

TURBO PASCAL

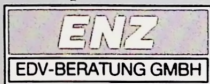
Requires Turbo Pascal 6.0, 5.5 or 5.0, or
Turbo Pascal for Windows.

Dual media included.

Version 5.23
S/N: 706056

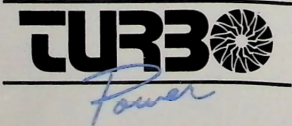
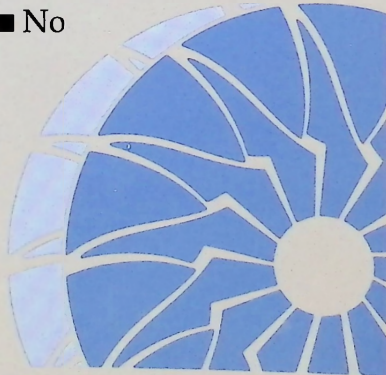
B-Tree Filer™

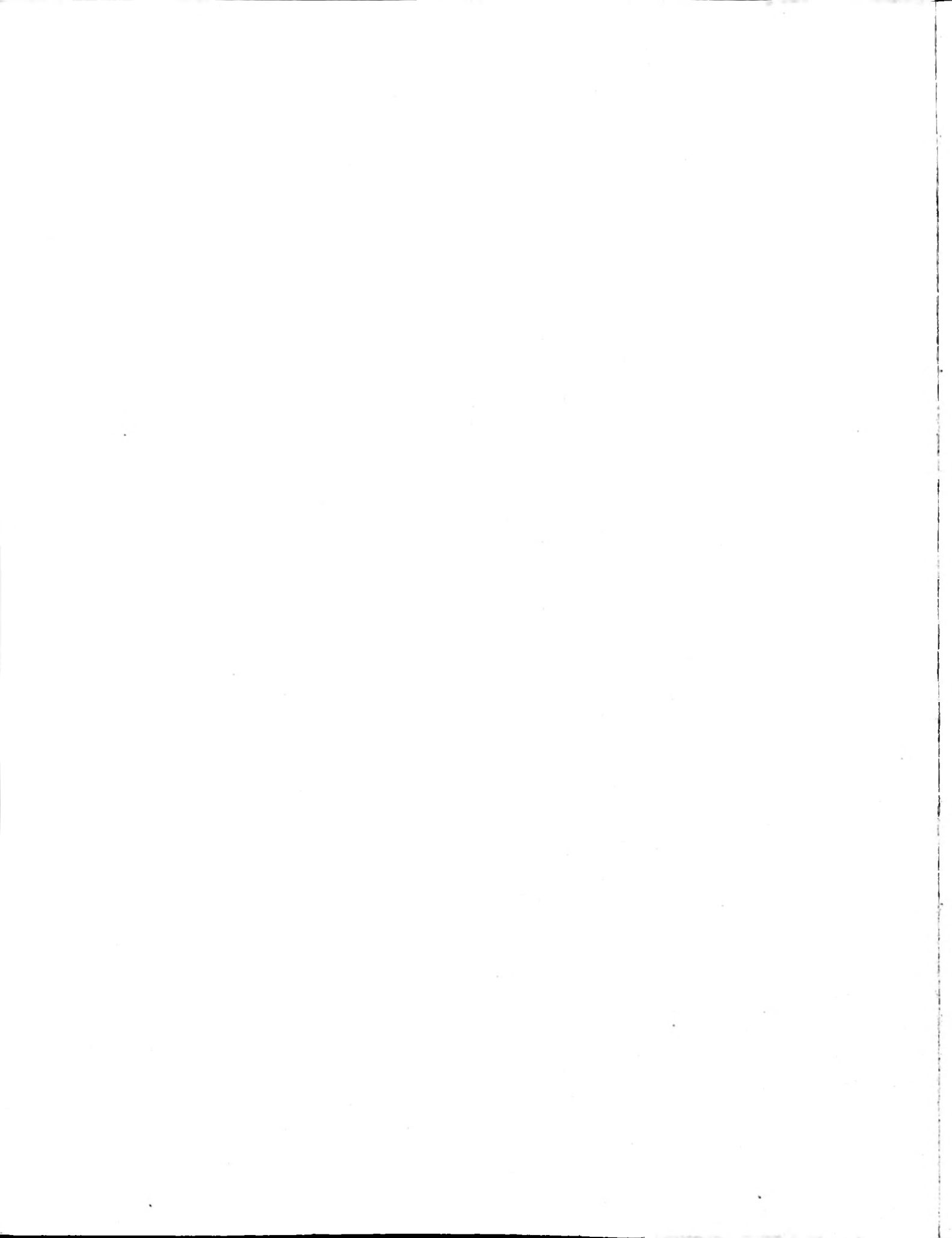
In cooperation with



Not for sale in Germany,
Switzerland or Austria.

Write powerful multi-user databases faster and easier entirely in Turbo Pascal. Easily upgrade from Borland's Database Toolbox. You get ■ B+tree indexing ■ Fixed and variable length records ■ Failsafe mode with journaling ■ Complete source code ■ Compatibility with Novell, Lantastic, MS-Net, Vines, and others ■ No royalties.





B-Tree Filer

User's Manual

**Copyright (c) 1989-1991, TurboPower Software
All Rights Reserved**

Third Edition December 1991

**P.O. Box 49009
Colorado Springs, CO 80949-9009**

**Technical Support: 719-260-6641
Sales: 800-333-4160
9 a.m. to 5 p.m. Mountain Time, Monday-Friday**

**Fax: 719-260-7151
CompuServe Mail: 76004,2611
CompuServe Support: PCVENB Section 6**

Trademarks Mentioned

TurboPower Software and the distinctive TurboPower logo are trademarks of TurboPower Software. B-Tree Filer is a trademark of TurboPower Software.

Turbo Professional is a registered trademark of Sunny Hill Software, used under license to TurboPower Software.

B-Tree ISAM is a trademark of Enz EDV-Beratung GmbH (West Germany).

Turbo Pascal, Turbo Assembler, Turbo Database Toolbox, and Turbo Editor Toolbox are trademarks or registered trademarks of Borland International.

MS-DOS, MS-NET, and Macro Assembler are trademarks of Microsoft Corporation.

IBM PC, XT, AT, PS/2, IBM PC LAN, and NetBIOS are trademarks or registered trademarks of International Business Machines Corporation.

Novell, NetWare, SFT, TTS, and Internetwork Packet Exchange (IPX) are trademarks or registered trademarks of Novell Inc.

Arcnet is a registered trademark of Datapoint Corporation.

3Com is a trademark of 3Com Corporation.

Network-OS is a trademark of CBIS, Inc.

PC-NET is a trademark of Orchid Technology, Inc.

PC-MOS/386 is a trademark of The Software Link, Inc.

Ethernet, Internetwork Packet Protocol, and Sequenced Packet Protocol are trademarks of Xerox Corporation.

Btrieve is a registered trademark of SoftCraft, Inc.

WordStar is a registered trademark of MicroPro International Corporation.

DesqView is a registered trademark of Quarterdeck Office Systems.

Table of Contents

1. Introduction	1
A. Requirements	3
B. Installation	5
C. NETDEMO Demonstration Program	10
D. Getting Help with POPHELP	15
E. Purchase Agreement	23
2. Configuration	25
A. Network Definition	26
B. Using EMSHEAP	29
C. Selection of Key Type	30
D. Other Conditional Defines	31
E. Compiler Options	33
F. Workstation Numbers	36
G. The Network Interface	40
3. Principles of Operation	43
A. Data and Keys	44
B. B-tree Organization	45
C. Managing Keys	50
4. Using B-Tree Filer	55
A. Fileblocks	55
B. Program Organization	57
C. Single User Example	58
D. Locking Techniques	69
E. Converting Single User Programs	72
F. Network Example	76
G. Questions and Answers	81
5. B-Tree Filer Identifiers	83

6. B-Tree Utilities	171
A. EMSHEAP: Using EMS for Heap Storage	173
B. VREC: Variable Length Records	197
C. BROWSER: Displaying a Fileblock	216
D. REBUILD/VREBUILD: Rebuilding a Fileblock	245
E. REORG/VREORG: Reorganizing a Fileblock	250
F. FIXTOVAR: Converting to Variable Length Records	254
G. NUMKEYS: Numeric Keys	257
H. ISAMTOOL: Miscellaneous Routines	266
I. MSORT: Sorting	273
 7. Network Utilities	 287
A. NETWARE: Novell NetWare	288
B. NETBIOS: NetBIOS Messages	374
C. SHARE: File Sharing	403
D. Network Demonstration Programs	418
 8. Appendix	 421
A. Error Codes	421
B. Procedure and Function Reference	426
C. Converting from Borland's Database Toolbox	441
D. Converting from B-Tree Filer 5.0	450
 Identifiers	 455
 Index	 461

1. Introduction

The storage and management of data is one of the central problems of electronic data handling. It wouldn't be an exaggeration to say that, for many programmers, database work is what pays the bills. Up to now, there haven't been many tools available to make writing database programs easier and more efficient in Turbo Pascal. That's where B-Tree Filer enters the picture.

B-Tree Filer offers you a high-quality library to manage databases in an efficient and easy-to-use manner. It includes many features that set it apart from the competition:

- integral support for multi-user, networked databases
- strict error checking and safety modes to maximize data integrity
- written in Turbo Pascal, with full source code provided
- fixed and variable length record support
- all indexes stored in one file to minimize file handle usage
- separate data and index files to maximize data integrity
- support for EMS and normal heap memory for index page buffers
- no TSRs required at runtime
- utility units for browsing, rebuilding, and sorting data files
- support for Novell, MS-NET, and all NetBIOS compatible networks
- network access units for file management, printer control, and message passing

B-Tree Filer is based on an improved version of the balanced B-tree algorithm, which has proven itself as the best method to access data in large databases. B-Tree Filer's capacity is so large that you should never need to worry about it:

- Maximum number of data records: greater than 2 billion
- Maximum number of key entries: greater than 2 billion
- Key length: 1 to 255, default value 30 characters maximum
- Maximum number of indexes per database: 254, default value 100
- Range for data record length: 21 bytes to greater than 2 gigabytes (limited by DOS and the 8086 architecture to 65,520 bytes)
- Maximum number of workstations: 65,534, default value 50

As you probably already know, B-Tree Filer is available in two versions. The single user version lets you write databases that run on individual PC's, while the multi-

6. B-Tree Utilities	171
A. EMSHEAP: Using EMS for Heap Storage	173
B. VREC: Variable Length Records	197
C. BROWSER: Displaying a Fileblock	216
D. REBUILD/VREBUILD: Rebuilding a Fileblock	245
E. REORG/VREORG: Reorganizing a Fileblock	250
F. FIXTOVAR: Converting to Variable Length Records	254
G. NUMKEYS: Numeric Keys	257
H. ISAMTOOL: Miscellaneous Routines	266
I. MSORT: Sorting	273
7. Network Utilities	287
A. NETWARE: Novell NetWare	288
B. NETBIOS: NetBIOS Messages	374
C. SHARE: File Sharing	403
D. Network Demonstration Programs	418
8. Appendix	421
A. Error Codes	421
B. Procedure and Function Reference	426
C. Converting from Borland's Database Toolbox	441
D. Converting from B-Tree Filer 5.0	450
Identifiers	455
Index	461

1. Introduction

The storage and management of data is one of the central problems of electronic data handling. It wouldn't be an exaggeration to say that, for many programmers, database work is what pays the bills. Up to now, there haven't been many tools available to make writing database programs easier and more efficient in Turbo Pascal. That's where B-Tree Filer enters the picture.

B-Tree Filer offers you a high-quality library to manage databases in an efficient and easy-to-use manner. It includes many features that set it apart from the competition:

- integral support for multi-user, networked databases
- strict error checking and safety modes to maximize data integrity
- written in Turbo Pascal, with full source code provided
- fixed and variable length record support
- all indexes stored in one file to minimize file handle usage
- separate data and index files to maximize data integrity
- support for EMS and normal heap memory for index page buffers
- no TSRs required at runtime
- utility units for browsing, rebuilding, and sorting data files
- support for Novell, MS-NET, and all NetBIOS compatible networks
- network access units for file management, printer control, and message passing

B-Tree Filer is based on an improved version of the balanced B-tree algorithm, which has proven itself as the best method to access data in large databases. B-Tree Filer's capacity is so large that you should never need to worry about it:

- Maximum number of data records: greater than 2 billion
- Maximum number of key entries: greater than 2 billion
- Key length: 1 to 255, default value 30 characters maximum
- Maximum number of indexes per database: 254, default value 100
- Range for data record length: 21 bytes to greater than 2 gigabytes (limited by DOS and the 8086 architecture to 65,520 bytes)
- Maximum number of workstations: 65,534, default value 50

As you probably already know, B-Tree Filer is available in two versions. The single user version lets you write databases that run on individual PC's, while the multi-

user version includes capabilities for record locking and network access. To differentiate the two versions in this manual, we'll use the phrase "B-Tree Net" whenever we refer to a capability that's specific to the network version, and "B-Tree Filer" when it's available in either version. Each section of the manual will make clear what capabilities each version provides. Note, however, that B-Tree Filer is designed so that you can write single user applications today that are easily upgradeable to network programs later.

B-Tree Filer is TurboPower Software's version of a solid and respected product from West Germany: B-Tree Isam. That product has been available for several years in Europe and has earned a well-deserved reputation for speed and dependability. Besides translating it into English, TurboPower Software has improved the original package by including the new virtual sort and network utilities. And we'll provide the same high level of service and support on which many have come to rely.

If you've upgraded to the current version of B-Tree Filer from version 5.0, be sure to read Appendix D. As a result of enhancements made in version 5.2, it is not source-compatible with earlier versions.

A. Requirements

To use B-Tree Filer, you must have the following:

1. An IBM PC, XT, AT (or close compatible), or PS/2 running DOS 2.0 or later. (When you are using network capabilities, the network itself may require DOS 3.1 or later.)
2. Turbo Pascal version 5.0, 5.5, or 6.0, or QuickPascal version 1.0.
3. Two floppy disk drives. A hard disk is desirable, however. Installation and compilation of all files will consume about 1.5 megabytes of disk space.
4. Required memory size is strongly influenced by configuration and usage of the provided units. The main FILER unit typically uses 64K to 128K bytes of memory for both code and data.

Optional:

5. Borland's Turbo Assembler (TASM) or Microsoft's Macro Assembler (MASM). Needed only if you wish to modify the assembly language source code for certain portions of B-Tree Filer.
6. Network hardware and operating system compatible with B-Tree Filer. These include Novell Netware, 3Com, PC-NET, PC LAN, MS-NET, Network-OS, PC-MOS/386, Alloy NTN, and others. Any network that supports DOS 3.1 record locking calls will run with B-Tree Filer.

Practically speaking, there are three network capabilities upon which B-Tree Net depends:

- the ability to define a disk storage area where files may be shared among various workstations
- the ability for a given workstation to lock part or all of a file, so that attempted access by another workstation leads to a detectable error
- the ability for each workstation to identify itself in some unique way

The third requirement can be simulated using software techniques, so it actually imposes no additional requirement on the network. If the network does not provide for a workstation identification number, however, the application's code will be slightly more complicated, as described in Section 2.F, "Workstation Numbers". B-Tree Net is not sensitive to the transport mechanism (e.g., Ethernet or Arcnet) used by the network, except insofar as it affects the ability to identify workstations.

It is relatively easy to add support for new networks to B-Tree Net. See Section 2.G.

There are also certain assumptions that this documentation makes of the programmer. It provides no tutorial background on Pascal programming; the user is expected to have an understanding of Turbo Pascal syntax, unit structure, and use of the compiler itself.

Although B-Tree Filer handles the details of B-tree management, the documentation assumes a basic understanding of record files and indexing techniques. Chapter 3 provides a brief explanation of B-tree data and index management. You should also have a basic understanding of DOS file access techniques and error codes.

The manual does not provide an in-depth tutorial on multi-user programming, although the discussions in Chapter 4 go some way towards explaining the issues and solutions. Neither does the documentation describe all of the background for accessing network protocols such as provided by NetBIOS and NetWare. Where possible, you should obtain the references mentioned in the appropriate chapters for further background. The manual does not describe how to install network operating systems; see your network documentation or system consultant for further information.

B. Installation

B-Tree Filer is distributed with the following 5.25" 360K diskettes:

FILER

The main files for the single user version of B-Tree Filer. Also contains various utilities for dearchiving the library, printing text files, and upgrading older Filer applications to the current version.

DEMO/HELP

Precompiled demonstration programs and an archive containing a popup help utility and files comprising a hypertext help database for B-Tree Filer.

NET (supplied only with the Network version)

Additional files needed for multi-user support in the FILER unit, as well as units for direct network access. Also contains precompiled network demonstration programs.

BONUS

Various extensions and enhancements to B-Tree Filer. These files are sometimes written by authors besides TurboPower and are not supported in the same way as the completely commercial portions of the product. See the READ.ME! file on the BONUS diskette for further information.

The B-Tree Filer distribution set also includes 3.5" diskettes. The FILER and DEMO/HELP diskettes are combined onto one 720K microfloppy, the NET diskette is on another (for purchasers of the network version only), and the BONUS diskette is on another.

Before using any of the diskettes, we recommend that you make backup copies. The files are not copy-protected.

We'd like to draw your attention to several files in particular, all of them found on the FILER diskette:

READ.ME

Please read this file before proceeding. It contains the latest updates to the printed documentation.

FASTUPD.xxx

Describes changes in this version of B-Tree Filer compared to the most recent previous one. This file is of particular use to Fast Update Plan subscribers. The characters "xxx" signify the current version number; for example, 522 means version 5.22.

LHARC.EXE

A freeware compression/decompression utility used to compress and decompress files with extension LZH. Type LHARC to get a summary of its options. LHARCMAN.LZH on the BONUS disk contains complete documentation of LHARC.

PACKING.LST

A complete list of all the files on all the diskettes, including the contents of the archives.

PRINTDOC.EXE

Prints text files with pagination, margins, and headers. This small utility is useful for printing READ.ME files and the like. Type PRINTDOC to get a list of options.

UPGRADE.EXE

A program that partially automates conversion of B-Tree Filer applications written for version 5.0 to this version. See Appendix D for a thorough discussion of conversion issues.

Installation of B-Tree Filer is so straightforward that we haven't provided a separate installation utility. We assume here that you have a hard disk with at least 1.5 megabytes of disk space free.

First, make a directory to contain all of the B-Tree Filer files. We recommend that you use the \BTREE subdirectory name and will refer to that name throughout the rest of this section. If you're updating an earlier version of B-Tree Filer, we recommend that you either create a new directory for this version or that you completely clear out the earlier directory. Filer 5.2 adds new files relative to version 5.0 and does not use others that existed in 5.0.

Next, copy all the files from the FILER, DEMO/HELP, and NET diskettes into the \BTREE directory. The diskettes have no subdirectories, so you can simply type, e.g., COPY A:*.*. You may also want to copy files from the BONUS diskette into the same directory; we recommend that you look at the BONUS diskette's READ.ME! file first and decide which bonus capabilities interest you.

Finally, set the current drive and directory to your \BTREE location, and enter the following two DOS commands:

```
LHARC X *.LZH  
DEL *.LZH
```

These commands dearchive the compressed files and then delete the archives themselves.

If you'll be working from another directory while using the B-Tree Filer units, be sure to add your \BTREE subdirectory to Turbo Pascal's search paths for Include Directories, Unit Directories, and Object Directories. If you use the command line compiler, you'd want to add the following lines to your TPC.CFG file:

```
-Ud:\btree  
-Od:\btree  
-Id:\btree
```

where d: specifies the drive where your \BTREE directory is located.

Note that we don't provide precompiled TPU files for any of the Filer units, since we're not sure which version of Turbo Pascal you're using and we also don't know how you'll configure B-Tree Filer. Chapter 2 goes into detail about configuration. In any case, the first time you use Filer units in your application, simply ask the compiler to Make your program and it will create the required Filer TPU files automatically.

We've also provided make files for use with Borland's standalone MAKE utility. There's no requirement for you to use these files; they're provided primarily as a complete specification of the interdependencies of the files, and also as a convenient way to build all the TPU and OBJ files at once if you feel the need. You should use FILER.MAK if you have the single user version of B-Tree Filer, or NETFILER.MAK if you have the network version. You may need to edit the make file in order to specify the location and version of your Turbo Pascal compiler. The make files contain instructions on how to perform any needed modifications. To run NETFILER.MAK, you'd type the following DOS command line (note that -f must be lowercase):

MAKE -fNETFILER

Section 1.D describes the popup help utility POPHELP.EXE, as well as the use of MAKEHELP.EXE to compile the supplied help text into an indexed, compressed file.

Here's a brief overview of the precompiled demonstration programs that you'll find in your \BTREE directory:

BIGSORT.EXE

Sorts text files, demonstrating the merge sort unit MSORT.

NETDEMO.EXE

A network-ready application that uses a scrolling fileblock browser to select records from the supplied ADDRESS database. Demonstrates code to add, delete, modify, search for, filter, and purge variable length records. See Section 1.C for more information.

NETINFO.EXE (Network version only)

Reports information about the current network, if any, using capabilities of the NETWARE, NETBIOS, and SHARE units. See Section 7.D for more information.

NSEND.EXE

NRECEIVE.EXE (Network version only)

Copy files from one workstation to another using NETBIOS or NETWARE capabilities. See Section 7.D for more information.

B-Tree Filer provides and documents the following units:

BROWSER

For displaying and browsing through B-Tree Filer fileblocks. See Section 6.C for more information. Also see FBROWSE.LZH on the BONUS diskette for a similar object-oriented unit that is compatible with Object Professional.

EMSHEAP

A heap manager for EMS memory. Used by the FILER unit to keep index page buffers in EMS. See Section 6.A.

FILER

The main unit for managing data and index files in a single-user or network setting. Chapters 2 through 5 describe this unit in detail.

FIXTOVAR

For converting fixed length record fileblocks to ones containing variable length records. See Section 6.F.

ISAMTOOL

Miscellaneous routines that extend the FILER unit. See Section 6.H.

ISCOMPAT

Compatibility routines that make the FILER unit of version 5.2 look more like the FILER unit of version 5.0. See Appendix D.

MSORT

A merge sort algorithm for sorting data sets of virtually unlimited size. See Section 6.I.

NETBIOS (Network version only)

For accessing NetBIOS-compatible network adapters, primarily for station to station communication. See Section 7.B.

NETWARE (Network version only)

For calling Novell's NetWare operating system services. See Section 7.A.

NUMKEYS

For converting various numeric types to and from strings, and for compressing strings used as index keys. See Section 6.G.

REBUILD

For purging deleted records and rebuilding indexes of a fileblock. See Section 6.D.

REORG

For changing the record structure of a fileblock or importing foreign data. See Section 6.E.

SHARE (Network version only)

For accessing various services of the DOS SHARE resident utility. See Section 7.C.

VRCOMPAT

Compatibility routines that make the VREC unit of version 5.2 look more like the VREC unit of version 5.0. See Appendix D.

VREBUILD

Like REBUILD, but for variable length record fileblocks. See Section 6.D.

VREC

For managing variable length records, in combination with the FILER unit. See Section 6.B.

VREORG

Like REORG, but for variable length record fileblocks. See Section 6.E.

C. NETDEMO Demonstration Program

To get a feel for B-Tree Filer's capabilities, you'll probably want to try out the NETDEMO program. The version of NETDEMO.EXE we provide is compiled using Filer's dynamic network capabilities; that is, it can run in single user mode or on any of the networks that B-Tree Filer itself supports.

NETDEMO has small data and index files ready for you to inspect or change. To do so, be sure that NETDEMO.EXE, ADDRESS.DAT and ADDRESS.IX are in the current directory. Then call NETDEMO with one of the following command line options:

/B	MS-Net compatible with NetBIOS machine name support
/C	CBIS' Network-OS
/D	Single-user DOS, no network
/M nnn	MS-Net or compatible
/N	Novell's Advanced NetWare
/P	Software Link's PC-MOS 386
/Q	DesqView
/V	Banyan's Vines
/X	Alloy's NTN

You can get a summary of these options (and any new ones) by just typing NETDEMO at the DOS command line. In the case of the /M option, you must also enter a unique workstation number for each instance of NETDEMO. See section 2.F for additional information.

NETDEMO shows off various capabilities of B-Tree Filer, including:

- single- or multi-user operation in the same program
- multiple key types in one index file
- save mode to guard data integrity
- variable length records
- database browsing, with scroll bar and mouse support and record filtering
- high-level facility to purge data files and rebuild indexes

NETDEMO.EXE also incorporates capabilities from TurboPower's Turbo Professional library--validated data entry screens and memo editing. In order to recompile NETDEMO.EXE, you'll need to have Turbo Professional version 5.02 or

later. If you don't have that, you can experiment with the supplied SIMPDEMO.PAS source file. It works just like NETDEMO, lacking only variable length records and validated entry. (Mouse support is available in SIMPDEMO only if you own Turbo Professional or Object Professional.) If you're looking for something like NETDEMO that works with the window hierarchy of Object Professional, see the FBROWSE.LZH archive on your B-Tree Filer BONUS disk.

NETDEMO manages an address file with the following fields:

```

FirstName : String[15];
LastName  : String[15];
Company   : String[25];
Address   : String[25];
City      : String[15];
State     : String[02];
Zip        : String[10];
Telephone : String[12];
Memolen   : Word;
Memo      : variable length buffer of 1-1200 bytes

```

Two index keys are used to find records quickly. The primary key (one that does not allow duplicates) is a combination of LastName and FirstName. The second key is the zip code, and it does allow duplicates.

The NETDEMO program lets you add, modify, or delete records, browse through a list of the records sorted by either key, search on any field, print the records, and rebuild the indexes after purging deleted records. NETDEMO uses a full-screen display to make all these operations easy.

B-Tree Filer Demo Program		Modify	Key: Last Name
Zip	Name	Modifying Record # 4	
02100	Dukakis, Michael	First name	[Philippe]
10016	Dvorak, John C.	Last name	[Kahn]
95066	Folks at, The	Company	[Borland International]
98073	Gates, Bill	Address	[1800 Green Hills Road]
67890	Goeshere, Your Name	City	[Scotts Valley]
60123	Jackson, Jesse	State	[CA]
95999	Jobs, Stephen	Zip	[95066-0001]
95066	Kahn, Philippe	Telephone	[408-438-8400]
95555	Lucas, George	Notes	[]
97520	Mace, Paul		
02142	Manzi, Jim		
12121	North, Oliver		
90403	Norton, Peter		
84057	Petersen, Paul		
03458	Pournelle, Jerry		
77071	Presley, Elvis		
11111	Quayle, J. Danforth		
23456	Swaggart, Jimmy		
30311	Turner, Ted		
56789	Tyson, Mike		
		Sure, Microsoft's bigger, but they don't have a horns section. Line: 1 Column: 1 7% Insert Indent Wrap	

<F2>Add <F3>Del <F4>Find <F5>Key <F6>Filt <F8>Prn <F9>Info <F10>Purge <Esc>Quit

NETDEMO will first prompt whether to operate in "Save Mode". In this mode, B-Tree Filer is able to recover from system or network crashes without losing data. This reliability comes at the expense of speed, but the difference will be negligible for interactive data entry.

NETDEMO attempts to open an existing data set in the files ADDRESS.DAT and ADDRESS.IX. If these are found, NETDEMO proceeds. Otherwise, it asks if you want to create a new (empty) data set. We have supplied a small set of addresses to allow easy exploration of NETDEMO.

If you are creating a new data set, NETDEMO will first direct you to the record entry screen. Enter the appropriate record fields, and then press <CtrlEnter> to exit the record editor. Remember that the key formed by combining LastName and FirstName must be unique among all the records in the data set. (Using a name as a primary key isn't common because of the relatively high probability of duplication. NETDEMO does it to avoid having to create an artificial "customer number" field or the like. If you encounter a duplicate key problem with NETDEMO, you'll need to artificially modify one of the names.)

After the data set has at least one record, NETDEMO will offer a full screen browse window. Here you can use the <Up> and <Down> arrow keys to browse among the records, the <Left> and <Right> arrows to scroll horizontally to see all the fields, and other cursor keys to reposition the highlight bar, which indicates the currently selected record. You can modify or delete the selected record using other NETDEMO commands.

Clicking with the left mouse button on a record moves the highlight bar to that record. Clicking again selects the record for modification. A vertical scroll bar indicates the relative position of the current record. Clicking on the scroll bar moves the highlight to the corresponding relative record position. Clicking on the up/down arrows associated with the scroll bar moves the highlight one row up or down.

While the browser is active, you'll see a menu bar at the bottom of the screen. This indicates many of the function keys you can press to activate various NETDEMO functions. It looks like this:

```
<F2>Add <F3>Del <F4>Find <F5>Key <F6>Filt <F8>Prn <F9>Info <F10>Purge <Esc>Quit
```

This menu bar also contains mouse hot spots: clicking over the command name executes the corresponding command. Clicking the right mouse button is interpreted as if the <Esc> key were pressed, and in many situations clicking the left button is like pressing <Enter>.

Here is a brief description of each command:

<F1> Modify

Edit the contents of the currently selected record. NETDEMO will reindex the record when you've finished editing. Remember that the record must retain a unique primary key. When you're done, press <CtrlEnter> to accept any changes, or <Esc> to discard them and retain the previous version. You can also modify a given record just by pressing <Enter> while the highlight bar is over it.

<F2> Add

Add a new record. Press <CtrlEnter> to accept the new entry, or <Esc> to discard it.

<F3> Delete

Delete the currently selected record. NETDEMO will offer the opportunity for you to confirm your choice.

<F4> Find

Search for a record. NETDEMO presents a full-screen search template where you can enter search parameters in any desired fields. NETDEMO's searching is not case-sensitive, and matches do not have to be complete. For example, entering "IB" in the company name field would match records like "IBM" or "Ibsen Baking Company". If the search fields you enter are part of NETDEMO's index fields (LastName and Zipcode), then NETDEMO will do a fast indexed search to find the initial match, possibly followed by a sequential search to qualify any other match fields. If NETDEMO does not find a complete match, it attempts to position the browse cursor on the closest approximate match.

<F5> Select Key

By default, NETDEMO presents records sorted by the primary key (LastName). Use the <F5> command to switch to the other key, which will display the records in zip code order. The currently active index is displayed on the top line of the screen.

<F6> Filter

Display only selected records. NETDEMO presents a filter template where you can enter search parameters for any desired fields. Only the records that match the non-empty fields will be displayed in the browse screen. For example, entering '9' in the zip code field would cause NETDEMO to display only those records whose zip codes start with 9. Press <F6> again to disable filtering. You'll note that when filtering is active, the scroll bar disappears. That's because the records displayed when filtering is enabled may be widely separated in the key sequence, or may cover just a narrow range of the entire sequence, and therefore the scroll bar probably won't have much meaning. Disabling the scroll bar in this situation is the application's choice; the BROWSER unit will continue to display the scroll bar unless you explicitly disable it.

<F8> Print

Print all the records. When you press <F8>, NETDEMO first displays a small menu. Here you can choose to print the records in LastName or Zipcode order, or to cancel the print request. If you proceed, NETDEMO writes all records (one line per record) to the default printer PRN. While NETDEMO is printing, you can press a key at any time to abort the print job.

<F9> Info

Information about the current address data set appears on the bottom line of the screen. This includes the total number of records in the data file, the number that have been deleted, the database mode (Save or Normal), and the current workstation number. Press any key to continue. (When variable length records are used, the record count is somewhat misleading. Instead of the number of logical records in the database, it is the number of fixed length record "sections". See Section 6.B for more information.)

<F10> Purge

Rebuild the data and index files using just the data file. The files are first closed, and then the data file is read in sequential order. Previously deleted records are purged from the data file, and the indexes are rebuilt from scratch.

<Esc> Quit

Use this command to quit back to DOS. NETDEMO will offer an opportunity for you to change your mind.

NETDEMO Design

NETDEMO uses a simple approach to data locking in a multi-user environment. When it needs to guarantee access to a particular record, NETDEMO simply locks the entire database (with <BTLockAllOpenFileBlocks>) for the shortest period of time possible. While reading records, it first checks for other locks on them. If a record is locked, NETDEMO prompts whether to try again, and aborts the operation if you request. (The actual locking process is more complicated than this superficial introduction. See Section 4.D for the full story.)

NETDEMO buffers the currently selected record in memory. This minimizes the amount of additional reading required when you modify a record. However, additional care is required when a modified record is to be written back to disk. Has the record been deleted in the meantime? Has someone else changed it? See Section 4.E and NETDEMO's Modify procedure for the techniques used to handle these possibilities.

NETDEMO also demonstrates the VREC, BROWSER, and VREBUILD units. These high level units make it easy to provide variable record lengths, full-screen record displays, and automatic rebuild functions.

D. Getting Help with POPHELP

POPHelp is a memory-resident program designed to provide easy-to-access, online help for the B-Tree Filer library. Since it is based on the TSR swapping capabilities of Object Professional, it occupies as little as 8K of DOS memory. POPHELP uses help files generated by the MAKEHELP utility, also provided with B-Tree Filer, so you're able to merge help files from different TurboPower products or to extend the help on your own.

Installing POPHELP

POPHelp can be installed in several ways:

- as a normal TSR using 150-200K of RAM
- as a TSR that swaps itself to EMS (expanded) memory, leaving only an 8K kernel behind in normal RAM
- as a TSR that swaps itself to disk, leaving an 8K kernel in RAM
- as a shell that execs another program and then automatically removes itself from memory.

You select among these options by using POPHELP command line parameters.

Supplied with this version of POPHELP is a help file for B-Tree Filer. This help file contains information about all interfaced constants, types, variables, procedures, and functions in the library. The help file is supplied in source format so that you can add to it as you see fit. Compiling it is a simple one-step operation.

The installation procedure described in Section 1.B will have stored the following files in your \BTREE directory:

```
POPHelp.EXE
MAKEHELP.EXE
BTFILER.TXT
*.TXT
```

BTFILER.TXT is the main help file, which references all of the other files with extension TXT. To compile the help text, enter the following at the DOS command line:

MAKEHELP /Q BTFILER

Once you've created the BTFILER.HLP file, you may delete the *.TXT files from your disk, unless of course you intend to modify them. You may also delete

MAKEHELP.EXE. This leaves two files related to help, POPHELP.EXE and BTFLILER.HLP, which together consume about 400K bytes of disk space.

This version of POPHELP is set up to load the BTFLILER.HLP file by default. Hence, in the simplest case, you make POPHELP memory resident by entering

POPHELP

at the DOS command line while BTFLILER.HLP is in the current directory. This will install POPHELP in swapping mode; it will use EMS memory (about 250K) or disk space (one or two files, each about 250K) for swapping and will retain about 8K of your DOS RAM.

POPHELP provides a number of options to control how it goes resident. It uses the following DOS command line syntax:

POPHELP [HelpFile] [Options]

HelpFile is an optional pathname to a help file. POPHELP will automatically load the BTFLILER.HLP file by searching in the current directory, in the directory of POPHELP.EXE (if you're running DOS 3.0 or later), and on the DOS PATH. Specify a different directory or a different help file by putting it on the POPHELP command line. POPHELP applies a default extension of HLP to the specified file name.

The following options may be specified on the command line:

/I	Use a single swap file (for RAM disks or XMS).
/A	Use normal file attribute (rather than hidden) for swap files.
/B	Force use of black and white video attributes.
/D	Force swapping to disk even if EMS is available.
/E ProgToExec CommandLine	Execute a program instead of going resident. No swapping. This option must be the last one on the command line.
/H Height	Specify initial help window height other than 12 rows.
/I HotKey	Specify alternate hot key for help index.
/L HotKey	Specify alternate hot key for screen lookup.
/M	Disable message appearing while TSR swaps to or from disk.
/N	Force entire TSR to remain in memory (no swapping).
/P HotKey	Specify alternate hot key for previous topic.
/S Path	Specify drive and directory for disk swapping.
/T	Display help text on second video monitor.
/U	Unload TSR from memory.
/V	Leave help text visible on second monitor after popping down.
/X	Use XMS memory for swapping, if available.
/?	Display a help screen.

The following command lines provide common examples of using the options.

POPHELP C:\BTREE\BTFILER

Load BTFILER.HLP from the specified directory.

POPHELP MYHELP /H 15

Load MYHELP.HLP using a default help window height of 15 rows.

POPHELP /D /M /S F:\1

Force POPHELP to swap to disk, using the root directory of drive F: for a single swap file. Disable swap messages. The single swap file is most often appropriate on RAM disks, which are very fast but have relatively limited storage space.

POPHELP /X /1

Enable POPHELP's use of XMS (extended) memory. Note that in order to use XMS you must load an XMS-compatible driver such as HIMEM.SYS, 386MAX.SYS, or QEMM386.SYS. The /1 option causes POPHELP to use half as much extended memory space, with slight degradation in performance.

POPHELP /E TURBO MYPROG.PAS

Load POPHELP temporarily and run the TURBO integrated environment.

POPHELP /N

Make POPHELP resident without swapping (keeps lots of DOS memory).

POPHELP /U

Unload the resident copy of POPHELP.

POPHELP /B

Force POPHELP to use only black and white (\$07, \$0F, \$70) video attributes.

POPHELP /T /V

Display help text on the second video adapter (the one that is currently inactive). When POPHELP exits, the help text will remain visible on the second display.

Keep in mind one important restriction when installing POPHELP in swapping mode. When POPHELP pops up, it disables any TSRs loaded chronologically after it. This is necessary because it probably overwrites their memory while it swaps in. Some TSRs, especially network shells and communication programs, won't accept being disabled in this way: the network will disconnect a station that doesn't respond

after a time, or a communication program will lose incoming characters. It's possible to avoid these problems by loading POPHELP after such TSRs. To avoid one common problem, POPHELP automatically detects when it is loaded before the Novell NetWare shell and refuses to pop up in this situation.

When the /E option is used, POPHELP is resident only for the time the specified program is running. When the program halts, POPHELP removes itself from memory automatically. POPHELP uses COMMAND.COM to run ProgToExec, so the program can be anywhere on the DOS PATH. You can also specify any command line parameters normally used by the program, with the exception that redirection of input or output is not allowed.

While POPHELP swaps to and from disk, it normally puts a status message on the top line of the screen. If you're swapping to a RAM disk, this status line may be an annoyance. If so, specify the /M option to disable the status.

POPHELP uses three different hot keys. By default, the hot keys are:

<LeftShift><F1>

Look up help topic based on word at cursor

<LeftShift><F2>

Redisplay previous help topic

<LeftShift><F3>

Display help index

The following options control the hot keys:

/L	Look up help topic
/P	Redisplay previous help topic
/I	Display help index

You can specify alternate hot keys at the time POPHELP is installed. On the command line, a hot key is specified as a hexadecimal word. The top byte specifies the shift key(s) to be pressed and may be zero. The bottom byte specifies the scan code for the hot key.

The shift key codes are:

No shifts	- 00
RightShift	- 01
LeftShift	- 02
Ctrl	- 04
Alt	- 08

Valid scan codes (in hexadecimal) are:

A - 1E	N - 31	0 - 0B	F1 - 3B	[- 1A
B - 30	O - 18	1 - 02	F2 - 3C	; - 27
C - 2E	P - 19	2 - 03	F3 - 3D	, - 33
D - 20	Q - 10	3 - 04	F4 - 3E	/ - 35
E - 12	R - 13	4 - 05	F5 - 3F	\ - 2B
F - 21	S - 1F	5 - 06	F6 - 40	] - 1B
G - 22	T - 14	6 - 07	F7 - 41	' - 28
H - 23	U - 16	7 - 08	F8 - 42	- - 34
I - 17	V - 2F	8 - 09	F9 - 43	` - 29
J - 24	W - 11	9 - 0A	F10- 44	
K - 25	X - 2D		F11- 57	
L - 26	Y - 15		F12- 58	
M - 32	Z - 2C			

For example:

/L 0244	performs lookup when <LeftShift><F10> is pressed
/P 0819	shows previous topic when <Alt><P> is pressed
/I 0517	displays help index when <Ctrl><RightShift><I> is pressed

Two hot keys may not be based on the same scan code, even if the shift keys differ. For example, you cannot use <Alt><F1> for one hot key and <Ctrl><F1> for another.

Using POPHELP

Once POPHELP is installed you can go about your business. The most likely place for you to start using it is within your text editor or within the TURBO integrated environment.

If you want to use a particular B-Tree Filer routine and you can remember its name but not its parameters, position the editor cursor within or immediately after the procedure name you've typed in. Then press the Lookup hot key, <LeftShift><F1> by default. POPHELP searches the list of topics looking for an exact match (disregarding case). If that fails, it looks for a topic whose name starts with the search string.

If POPHELP can find a matching topic with one of these two search strategies, it will pop up a help window showing you the details about that routine. If it can't, it will instead pop up a window with the names of all the B-Tree Filer units.

If you're not sure of the name of a routine, but you remember which unit it's in, press the Index hot key, <LeftShift><F3> by default, and you'll get a pick list showing all the B-Tree Filer units. Select the unit you want and a list of topics within that unit will appear.

Within the help window you can use the arrow keys to move the cursor to highlighted areas which indicate links to other topics. Once the cursor is positioned within a highlighted region of interest, press <Enter> to bring up that topic. In this way you can browse around the help database at will. Note that the <Tab> and <ShiftTab> keys make the cursor jump immediately from link to link.

Once you've seen the information you want, you can return to the underlying application by pressing <Esc>.

If you want to see the same help topic again, press the Previous Topic hot key, <LeftShift><F2> by default.

Pasting Help Text into your Editor

Among its other capabilities, POPHELP also can paste marked blocks of help text into your editor. Cut/paste works much the same way that block copying works in the TURBO integrated environment.

The first step is to position your editor's cursor at the exact location where you want to paste the help text. Next, pop up the help screen and select your topic. Move the cursor through the text until it is positioned on the first character that you want to paste. Mark the beginning of the block by pressing <F7> (or <CtrlK>). Scroll through the text again until the cursor is positioned just beyond the last character that you want to paste. Press <F8> (or <CtrlK><K>) to mark the end of the block. POPHELP will highlight the block. Finally, press <F4> (or <CtrlK><C>) to copy the marked block into your editor. POPHELP will immediately pop down and start feeding keystrokes to the underlying application. The next time you invoke POPHELP, the block will be hidden. If you want to toggle the block's visibility at any time, press <CtrlK><H>.

If the pasted text causes your editor to scroll, the display may temporarily look pretty strange since POPHELP can type much faster than the editor was probably designed to accept. However, your editor will "catch up" when POPHELP finishes pasting characters.

If your editor has an automatic indentation feature you will probably want to disable it--e.g., <CtrlQ><I> toggles autoindent in the TURBO IDE--during the paste. If autoindent is on during pasting, the pasted text may gradually move to the right of the screen.

If the pasted block is larger than POPHELP's internal paste buffer (512 keystrokes), POPHELP must swap itself into memory to refill the buffer each time it is exhausted. When this occurs the pasting will be temporarily delayed until the buffer is refilled and POPHELP swaps out again. Depending on the speed of your system

and whether POPHELP swaps to EMS, XMS or disk, this delay may range from a fraction of a second to a few seconds. If POPHELP cannot safely swap itself in when the buffer empties, the paste will be aborted prematurely. This should not occur when POPHELP is used over modern programmer's editors; EDLIN may have trouble.

POPHelp Command Summary

The following table summarizes the POPHELP commands we've discussed so far as well as a few commands we haven't mentioned yet.

<F1>

Displays the master topic index.

<AltF1>

Displays the topic that was displayed just prior to the current one. The cursor is positioned to its last location in the help window.

<F3>

Prompts for a new help file. A default extension of HLP is added to the name you enter. POPHELP will read any help file compiled using the MAKEHELP utility provided with B-Tree Filer or Object Professional. Note that each help file needs memory space that is proportional to the help file's size and number of topics. POPHELP reserves a certain amount of memory when it goes resident and it cannot increase that amount later. Therefore, if you expect to load multiple help files during one POPHELP session, you should load the largest of them when POPHELP first goes resident.

<F4>, <CtrlK><C>

If a block is marked and visible, pastes it from the help system to the underlying editor.

<F5>, <AltZ>

Toggles the zoom mode of the window.

<F6>, <AltM>

Enters move/resize mode--you can use the cursor keys to move the window around the screen, or the Shift-cursor keys to resize the window. In move/resize mode you can also perform "speed resizing" by pressing <CtrlLeft> or <CtrlRight> for horizontal changes and <PgUp> or <PgDn> for vertical changes. Note that the window cannot be resized while it is zoomed.

<F7>, <CtrlK>

Marks the beginning of a block.

<F8>, <CtrlK><K>

Marks the end of a block.

<F9>, <CtrlL>

Continues the last search (initiated by the <LeftShift><F1> hotkey). Searching starts with the next topic beyond the current one and is circular, wrapping from the last topic to topic 1 as many times as you like.

<CtrlK><H>

Hides or unhides the currently marked block. <F4> will paste only if a block is visible.

<PgUp>, <PgDn>

Moves to the next or previous page of the current help topic.

<Left>, <Right>, <Up>, <Down>

Moves the cursor around the help screen. Attempting to move the cursor beyond the edge of the window will cause more text to scroll into the window.

<Tab>

Jumps to the next cross reference topic, if any.

<ShiftTab>

Jumps to the previous cross reference topic, if any.

<Enter>

Selects the cross-reference topic at the cursor location, if any. The new topic is loaded into the window and the cursor positioned on the topic's first character.

<Esc>, <AltX>

Exits POPHELP. The current topic and page are retained and may be redisplayed by pressing the Previous Topic hot key, <LeftShift><F2> by default.

E. Purchase Agreement

This software and accompanying documentation are protected by United States copyright law and also by International Treaty provisions. Any use of this software in violation of copyright law or the terms of this agreement will be prosecuted to the best of our ability.

Portions of the software are copyright (c) 1986-1991 by Dipl.Math. Ralf Nagel and copyright (c) 1989-1991 by Enz EDV-Beratung GmbH, both of West Germany. Other portions of the software are copyright (c) 1989-1991 by TurboPower Software. TurboPower Software distributes the copyrighted works under exclusive license from Enz, who can be reached at the following address:

Enz EDV-Beratung GmbH
Wetterauer Str. 3
W-6380 Bad Homburg 6
Germany

TurboPower Software authorizes you to make archival copies of this software for the sole purpose of back-up and protecting your investment from loss. Under no circumstances may you copy this software or documentation for the purposes of distribution to others. Under no circumstances may you remove the copyright notices made part of the software or documentation.

You may distribute, without runtime fees or further licenses, your own compiled programs based on any of the source code of B-Tree Filer. You may not distribute any of the B-Tree Filer source code, compiled units, or compiled example programs without written permission from TurboPower Software.

Note that the previous restrictions do not prohibit you from distributing your own source code or units that depend upon B-Tree Filer. However, others who receive your source code or units would need to purchase their own copies of B-Tree Filer in order to compile the source code or to write programs that use your units.

The supplied software may be used by one person on as many computer systems as that person uses. We expect that group programming projects making use of this software will purchase an individual copy of the software and documentation for each member of the group. Contact TurboPower Software for volume discounts and site licensing agreements.

With respect to the physical diskettes and documentation provided with B-Tree Filer, TurboPower Software warrants the same to be free of defects in materials and

workmanship for a period of 30 days from the date of receipt. If you notify us of such a defect within the warranty period, TurboPower Software will replace the defective diskette(s) or documentation at no cost to you.

TurboPower Software warrants that the software will function as described in this documentation for a period of 30 days from receipt. If you encounter a bug or deficiency, we will require a problem report detailed enough to allow us to find and fix the problem. If you properly notify us of such a software problem within the warranty period, TurboPower Software will update the defective software at no cost to you.

TurboPower Software further warrants that the purchaser will remain fully satisfied with the product for a period of 30 days from receipt. If you are dissatisfied for any reason, and TurboPower Software cannot correct the problem, contact the party from whom the software was purchased for a return authorization. If you purchased the product directly from TurboPower Software, we will refund the full purchase price of the software (not including shipping costs) upon receipt of the original program diskette(s) and documentation in undamaged condition. TurboPower Software honors returns from authorized dealers, but cannot offer direct refunds to anyone who did not purchase a product directly from us.

TurboPower Software does not assume any liability for the use of B-Tree Filer beyond the original purchase price of the software. **IN NO EVENT WILL TURBOPOWER SOFTWARE BE LIABLE TO YOU FOR ADDITIONAL DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THESE PROGRAMS, EVEN IF TURBOPOWER SOFTWARE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.**

By using this software, you agree to the terms of this section. If you do not agree, you should immediately return the entire B-Tree Filer package for a refund.

TurboPower Software offers telephone support for B-Tree Filer on an as-available basis at no extra charge. You are welcome to call our support line on Monday-Friday 9am to 5pm (Mountain time) to ask for assistance. We also maintain a CompuServe forum area to support you. The account and telephone numbers are listed on the title page of this manual.

We believe in supporting our customers and will do our best to make the software perform the job it was designed to do!

2. Configuration

Before using B-Tree Filer's units, you need to understand and perhaps modify various settings that fall into the following categories:

- conditional compilation defines within BTDEFINE.INC
- other compiler options
- workstation numbers for network database applications
- the FILER network interface for non-supported networks

These configuration options control primarily the operation of the FILER unit, which is the core of B-Tree Filer's data management capabilities. The remainder of this chapter describes these topics in detail.

Many of the settings described here are controlled by conditional defines in BTDEFINE.INC. This file is a Pascal include file that is included into all B-Tree Filer units. To modify it, load it into a text editor and find the appropriate conditional symbol. TurboPower uses the convention of inserting a period between the '{' and the '\$' of a conditional define in order to deactivate that define. For example, the following define is active

```
{ $DEFINE NoNet }
```

The deactivated form of the same define is

```
{ . $DEFINE NoNet }
```

With the period in place, the compiler sees the line as a plain comment which has no effect on compilation.

After modifying BTDEFINE.INC, save it to disk and use the compiler to Make your application for the changes to take effect.

A. Network Definition

If you've purchased the network version of B-Tree Filer and you want to write network applications, you will need to configure the FILER unit before you can correctly access files on a network file server. Conditional defines located in BTDEFINE.INC specify which networks are supported and you must activate the desired defines for proper network support. This section of BTDEFINE.INC looks like the following:

```
{ $DEFINE NoNet }
{ . $DEFINE Novell }
{ . $DEFINE MsNetMachName }
{ . $DEFINE MsNet }
{ . $DEFINE CBISNet }
{ . $DEFINE PCMOS386 }
{ . $DEFINE VinesNet }
{ . $DEFINE NTNXNet }
{ . $DEFINE DesqView }
{ Valid network interfaces. One or more may be defined, but NoNet may not be
  selected except by itself. }
```

Since NoNet is defined, no network support code is compiled into the FILER unit. To activate one or more network interfaces, insert a period in front of the NoNet \$DEFINE, and remove the period from whatever networks you wish to support. Be sure that the compiler rebuilds FILER.TPU so that the changes take effect. Once one or more network interfaces are available, the <BTInitIsam> function is used to select which one to use for a given program run.

The following paragraphs offer more detail on each of the supported network options.

NoNet

Locking and file sharing routines do nothing when FILER is compiled with this define. Be very careful not to run a NoNet application in a multi-user setting. Many networks will let you proceed, but the data and index files are likely to become corrupted.

Novell

Runs on all networks compatible with Novell NetWare. This includes NetWare, Advanced NetWare, ELS (Entry Level) NetWare, and NetWare 386 systems. Uses Novell-proprietary record locking for optimized performance.

MsNet

Runs on any network compatible with Microsoft's locking calls, which were first implemented in the SHARE program distributed with MS-DOS 3.1. Uses DOS function call 5Ch for record locking. Since Microsoft does not define a generic

mechanism for identifying workstations, this define does not provide support for determining the workstation number. See Section 2.F, "WorkStation Numbers", for additional information.

Most networks (including Novell) offer support for Microsoft record locking, so MsNet is a good value to try if you're not sure which define is correct for your situation. However, if you have a network that implements both DOS record locking and NetBIOS support, it's likely that the MsNetMachName interface is a better choice.

MsNetMachName

Provides routines identical to those of MsNet, with one exception. Workstation identification is supplied via DOS function call 5Eh, supported by most NetBIOS emulators. You should use this define for the 3Com, PC-NET, MS-NET, PC LAN, and Lantastic networks.

CBISNet

Supports the Network-OS system from CBIS. This is a hybrid of MsNet and Novell, using the locking calls of MsNet and the workstation identification routine of Novell. This define requires a Network-OS version between 6.20 and 7.19, inclusive. Network-OS versions 7.20 and later no longer support the NetWare GetConnectionNumber call used by the CBISNet define. If you have such a recent version of CBISNet, use the MsNetMachName or MsNet defines instead.

PCMOS386

Supports the PC-MOS/386 operating system (version 2.01 or later) from The Software Link, Inc. in a multi-user environment. Record locking uses SHARE compatible calls, but workstation identification is implemented with a method specific to PC-MOS/386. The workstation number is equal to the "task number", plus one. See function <BTSetDosRetry> in Chapter 5 for additional information.

VinesNet

Supports the Vines Network from Banyan, Inc. (version 3.01(0) or later). This network interface is very similar to MsNetMachName, with only minor differences in the way the workstation number is computed. See Section 2.F, "WorkStation Numbers", for additional information.

NTNXNet

Supports the NTNX Network from Alloy. Another MsNet derivative, this network obtains a workstation number by making an int 7Fh call.

DesqView

Although DesqView isn't a multi-user operating system, many programmers use it as a development environment and as an inexpensive means to test multi-user software. The DOS SHARE utility must be loaded before DesqView is run. Record locking uses SHARE compatible calls, and the DesqView task number provides "workstation" identification.

B. Using EMSHEAP

Two defines within BTDEFINE.INC control whether the FILER unit can use EMS memory for storage of index page buffers. As described later, index buffers consume a minimum of 20K bytes of heap space using the default constants.

Larger buffers can significantly improve the performance of index-intensive operations, especially for single-user and lightly loaded multi-user applications. By placing index buffers in EMS memory instead of on the normal heap, you can reduce normal heap usage to a negligible amount and, in many cases, arrange to load the entire index into EMS RAM for the ultimate in performance.

To enable EMS support in Filer, activate the UseEMSHeap define in BTDEFINE.INC. This define is activated by default and pulls in about 5.5K bytes of code from the EMSHEAP unit. As described in Section 6.A, the EMSHEAP unit automatically detects EMS and initializes itself accordingly. The <BTInitIsam> procedure described in Chapter 5 is used to specify the amount of EMS to use, if any, for a particular run of a program.

The EMSDisturbance symbol, also in BTDEFINE.INC, may be defined when UseEMSHeap is defined. EMSDisturbance, which is off by default, controls whether the FILER unit restores its own EMS page frame context before it accesses the EMS page frame. (See Section 6.A for definitions and background.) This step is needed only if another portion of the application also uses EMS and modifies the page frame without saving and restoring its state. Note that Borland's OVERLAY unit, which is probably the most likely additional user of EMS, does save the page frame and therefore does not require EMSDisturbance to be defined. On the other hand, the EMS large arrays in Turbo Professional and Object Professional assume that they "own the page frame" and modify it without regard for any other code. Hence, for these units to coexist with the FILER unit, EMSDisturbance must be defined. Note that FILER always saves and restores the page frame context for the benefit of other users of EMS, regardless of the setting of EMSDisturbance.

Several other settings in BTDEFINE.INC control the EMSHEAP unit, and therefore they will indirectly affect FILER if UseEMSHeap is defined. These defines, UseTPEMS, UseOPEMS, DebugEMSHeap, NoErrorCheckEMSHeap, and ManualInitEMSHeap, are described in Section 6.A of this manual.

C. Selection of Key Type

All keys in a B-Tree Filer index are always stored as strings. The `FILER` unit can store either of two kinds of strings, however. If the `LengthByteKeys` define is active within `BTDEFINE.INC`, as it is by default, Turbo Pascal style strings are used: each string is composed of a length byte followed by the characters that comprise the string. If the `ASCIIZeroKeys` define is active, C style strings are used: the characters of the string come first and are terminated by a single null character (`#0`). Only one of the two defines may be active at one time.

The `ASCIIZeroKeys` define controls only the format of the strings stored in the index file. All strings passed to and returned from `FILER` routines continue to use the Pascal format. The purpose of this define is to create an index file that could also be read and written by a C version of B-Tree Filer.

If you change these defines, you must rebuild the index file of all affected fileblocks.

D. Other Conditional Defines

Several other defines in BTDEFINE.INC affect the FILER unit and other B-Tree Filer units.

InitAllUnits, which is not defined by default, controls whether each B-Tree Filer unit has an initialization block, even if only an empty one. Defining this symbol works around a bug in some early versions of Borland's Turbo Debugger. Defining it also generates 16 bytes of extra code for units that wouldn't otherwise have an initialization block, and generates a tiny delay at program startup for calls to the empty initialization blocks. You may need this define if when you run Turbo Debugger the source code does not appear in the module window for units you're sure you've compiled with debug information.

UseTPCRT and UseOPCRT determine whether the BROWSER unit takes advantage of the CRT and mouse functions of the units from Turbo Professional and Object Professional. If neither define is active, BROWSER uses Borland's CRT unit and mouse operations are not available in the browser. If your program already uses TPCRT and/or TPMOUSE, define UseTPCRT; if it already uses OPCRT and/or OPMOUSE, define UseOPCRT. Failure to do so will lead to a "CRT/TPCRT conflict" or at least the presence of wasted code in the application's EXE file. Note that if you enable UseTPCRT or UseOPCRT, you'll need to have a copy of TPDEFINE.INC or OPDEFINE.INC available if you compile BROWSER.PAS, SIMPDemo.PAS, or NETDemo.PAS. These include files are provided with Turbo Professional and Object Professional, respectively, but you may have deleted them after compiling the TPU files for those packages.

LockBeforeRead, which is not defined by default, offers an automatic workaround for a bug that appears in some versions of Novell's NetWare shell. This bug affects most shells with version numbers of 3.0x or 3.1x, which are shipped with NetWare 2.15C, 2.2, and 3.1. Under these versions, if one station places a record lock and a second station attempts to read the locked record, the second station will "hang" until the record lock is removed. The correct behavior, as implemented by other versions of NetWare, is for the second station to return immediately with a lock error. Novell's version 3.22 of the shell corrects this error. Older shells such as 2.12 also do not have the problem.

This bug affects B-Tree Filer applications only when record locking is used. By defining LockBeforeRead, you cause Filer to place a lock on any region of a file it is about to read, then to remove the lock when it is done. This prevents the "hang" since the initial lock will fail if a lock is already in place. Naturally, this behavior causes some negative effect on performance. Note that it is acceptable to activate

LockBeforeRead even if your network does not exhibit the erroneous NetWare behavior.

The symbol BTree52 is always defined in BTDEFINE.INC to identify the FILER unit naming conventions new to versions 5.21 and later. This define allows units that extend B-Tree Filer to be written to work with both the old and new calling sequences.

You should never modify the Heap6 define in BTDEFINE.INC, but you should understand its purpose. Heap6 is automatically defined when BTDEFINE.INC is compiled using the Turbo Pascal 6 compiler (and possibly will be for future compiler versions as well). Because Borland rewrote the compiler's heap manager for version 6, it's necessary to adjust various internal functions of B-Tree Filer to work with the differing heap managers. Alternate code is controlled via the Heap6 define.

Because B-Tree Filer no longer supports Turbo Pascal 4.0, a test in BTDEFINE.INC halts compilation if this version of the compiler is detected.

E. Compiler Options

Each unit provided with B-Tree Filer begins with source code similar to the following:

```
{ $I BTDEFINE.INC }
{ $I ISAMOPT.INC }
{ $I -,R- }
```

BTDEFINE.INC is included to allow specification of compilation directives. Note that you may modify BTDEFINE.INC to specify compiler options as well as just the conditional symbols described already. ISAMOPT.INC specifies standard compiler options, including the overlay directive \$O+. Finally, the unit source code continues with a group of directives that must be used for the unit; these settings override any that you've placed in BTDEFINE.INC or ISAMOPT.INC.

The following B-Tree Filer units may be overlaid given the default compiler directives in their source files:

```
BROWSER*, EMSHEAP*, EMSSUPP, FILER*, FIXTOVAR, ISAMTOOL*, ISCOMPAT, MSORT*,
NETBIOS, NETWARE*, NUMKEYS, REBUILD, REORG, SHARE*, TPALLOC, VRCOMPAT*,
VREBUILD, VREC*, VREORG
```

Units whose names are followed by asterisks have initialization blocks. Overlaying such a unit requires that Turbo Pascal's overlay manager be initialized before the initialization block is executed. See the Turbo Pascal manual for more information.

Although almost any B-Tree Filer unit can be overlaid, performance considerations will often make it undesirable to do so.

Whenever you use library packages such as B-Tree Filer, it's important to remember that the Turbo Pascal compiler has a finite capacity. When you write a short program that uses units provided by others, you may forget that the compiler is actually processing lots of code provided by the units listed in the uses statement. To maximize the compiler's capacity, you should apply the following advice.

1. In the TURBO.EXE integrated environment, apply the following settings:

Compile/Destination	Disk
Options/Compiler/Debug Information	Off
Options/Compiler/Local Symbols	Off
Options/Linker/Map File	Off
Options/Linker/Link Buffer	Disk
Options/Debugger/Integrated	Off
Options/Debugger/Standalone	Off

The latter two options are in a slightly different menu location in versions of Turbo Pascal prior to 6.0.

Although these settings may lead you to believe that you can't use a debugger, that's not true. The important goal is to add debug information only to those units where you need to trace, not in every unit.

2. If you outgrow the capacity of the integrated environment, use the command line compiler TPC.EXE instead. When using TPC, make the following options part of your TPC.CFG file (see the Turbo Pascal manual) or part of your command line:

`/SD- /$L- /L /M`

These options turn off debug information, force link buffering to disk, and perform a make by default. Again, the key to compiling with debug information is to add such information to only those units where you need it and to strip the information after you're done with it. To compile a single unit with debug information, use the following command line:

`TPC /$D+ /$L+ /L UnitName`

To compile a program with debug information for Turbo Debugger, use the following command line:

`TPC /$D+ /$L+ /L /V ProgName`

Do not use the /V option regularly, since it causes the compiler to append debug information to your program's EXE file, which will bloat it significantly.

3. If you outgrow even the capacity of TPC.EXE, consider getting the TPCX.EXE protected mode compiler, which is provided with Borland's Turbo Pascal 6.0 Professional Edition. This version of the compiler uses up to 15 megabytes of extended memory to provide almost unlimited capacity.

There are various other tricks for improving compile capacity, although in most cases they offer second-order improvements compared to the items already listed. Keep the following in mind:

1. Unload as many TSRs and device drivers as possible.
2. Turn off range and stack checking with `{$R-,S-}`.
3. Extract all units except PRINTER.TPU from TURBO.TPL by using TPUMOVER.
Under Turbo Pascal 6, you can extract *all* units from TURBO.TPL and instruct the IDE not to look for TURBO.TPL at all. In this case, the compiler will load other units such as SYSTEM.TPU and DOS.TPU only

when the program calls for them.

4. Keep the main program as small as possible and instead put all code into units. After following this advice your main program will use a single main unit and contain a single call to the one "Main" routine interfaced by that unit.
5. Minimize the number of identifiers interfaced by every unit.
6. Carefully set the defines in files such as BTDEFINE.INC to activate only the library features that you need.

F. Workstation Numbers

B-Tree Filer requires that each workstation participating in a multi-user database application have a unique identifying number. This allows B-Tree Filer to manage the file locks applied to a fileblock. Since some networks do not provide an automatic means to obtain a unique workstation identifier, we must develop independent methods to do so.

The FILER unit interfaces three identifiers that are relevant to this discussion. <IsamWSNr> is a Word variable containing the number assigned to the current workstation. <MaxNrOfWorkstations> is a constant that defines the maximum number of workstations FILER can manage (default 50). <IsamWSNr> must be in the range from 1 to <MaxNrOfWorkstations>.

The <BTInitIsam> function automatically computes the value of <IsamWSNr> by calling an appropriate network service whenever this is possible. When it cannot, the application must assign a value to <IsamWSNr> before calling <BTInitIsam>, which then validates that the number falls in the required range. <BTInitIsam> cannot validate the uniqueness of the workstation number. If <BTInitIsam> detects an invalid number, it generates <IsamError> 10310.

There are several approaches to defining a unique workstation number:

1. Read it directly via a network service.
2. Construct it from the network name of each workstation. (Network names are available on most networks supporting the NetBIOS protocol.)
3. Hard code it into each workstation's copy of the application.
4. Read it from the workstation's DOS environment.
5. Obtain it from the DOS command line.
6. Use a shared file to log workstation numbers in and out dynamically.

Each solution is discussed in this section.

The first approach--reading the workstation number via a network service--is obviously the ideal. Novell and PC-MOS/386 networks, among others, provide this service. When one of these networks is selected, the other techniques described in this section are not required. B-Tree Filer calls the appropriate routines automatically and assigns <IsamWSNr> within <BTInitNetIsam>.

The second alternative--constructing a workstation number from the network name--is almost as good. The MsNetMachName and VinesNet network interfaces

use this approach. However, in order to generate a workstation number, the routines must convert a text string (returned by the NetBIOS from a name table constructed by the system administrator) to a unique integer in the range from 1 to <MaxNrOfWorkStations>. The FILER unit assumes that all machine names end with unique identifying digits and extracts the identifying digits in a straightforward manner. For example,

Machine Name	IsamWSNr
Hughes03	3
Sadowsky9	9
Server39	39
MT23R	error (doesn't end in digit)
Server	error (no digits at all)
X9999	error (station number too large)

In the last three cases, <BtInitIsam> would generate error 10310. It is the system administrator's responsibility to set up machine names that end with unique digits using a utility provided with the network.

Under VinesNet, the steps required to get valid workstation numbers using the machine name approach are somewhat more complicated. First, the NetBIOS driver must be installed and loaded. Then, each user profile must contain the following command:

```
SETNETB /N:"<Name>0"
```

where <Name> is any text string, subject to the following conditions:

- all user profiles on all workstations must contain the same name
- the length is limited to 10 characters
- the name may not end with a number but must be followed by a zero
- the name and following zero must be surrounded by quotes as shown

Hence, valid calls to SETNETB include

```
SETNETB /N:"HUGHES0"
SETNETB /N:"0"
SETNETB /N:"VinesNet0"
SETNETB /N:"Ver3.01(0)0"
```

Invalid examples include

SETNETB /N:"HUGHES30"	(ends with two numbers)
SETNETB /N:"VinesNet"	(doesn't end with zero)
SETNETB /N:"VinesNetVersion3.01(0)0"	(too long)

FILER's VinesNet network interface takes advantage of the fact that Vines automatically appends a unique identifying digit when it detects logins under duplicate NetBIOS names. The resulting sequence of names returned to FILER is therefore

<Name>0	first login
<Name>0.1	second login
<Name>0.2	third login

FILER extracts the digits at the end of each name, converts them to a number, and adds one to get a valid <IsamWSNr>.

If this naming convention cannot be used on a particular Vines installation, define and use FILER's MsNet interface instead and determine the workstation number with one of the other techniques described below.

The third alternative--coding the number into each copy of the application--is easy, but cumbersome and perhaps impractical. When it comes time for a new version, multiple unique copies must be generated and distributed. And this approach defeats one of the advantages of networking in the first place: storing and running a single copy of the application on the server.

The fourth alternative--reading the number from the workstation's DOS environment--is a sound approach provided that network users can be trusted to initialize their environment correctly, or at least leave it well enough alone. For example, the system administrator might require that each user put a line like the following in the AUTOEXEC.BAT file:

```
SET FILERWSNR=001
```

The DOS unit provided with Turbo Pascal 5.0 exports a routine to read environment strings. With it, we might construct a routine to build the workstation number as follows:

```
function WSNrFromEnvironment : Word;
var
  N : Integer;
  Code : Word;
  S : String;
begin
  S := GetEnv('FILERWSNR');
  if Length(S) = 0 then
    (Environment string wasn't found)
    WSNrFromEnvironment := 0
  else begin
    (Convert string to number)
    Val(S, N, Code);
```

```

    if Code <> 0 then
      (Environment string wasn't a valid number)
      WSNrFromEnvironment := 0
    else
      WSNrFromEnvironment := N;
    end;
  end;
end;

```

A return value of 0 is equivalent to reporting an error.

The fifth alternative--requiring that the user provide the number as a command line parameter--is also workable, but requires that the user "get it right" more often. Implementing this alternative is straightforward. See NETDEMO.PAS, which uses this technique, for details. It is essential here that the application check <IsamError> to assure that the user has entered a workstation number.

The final alternative--using a shared file to log workstation numbers dynamically--is a good generic solution. Once the program is set up correctly, workstation number determination does not depend on the user. We've provided supporting routines in the ISAMTOOL unit so that you don't need to invent them yourself. The login routine is declared as follows:

```

procedure WSNrLogIn(FName : IsamFileName);

```

Here's an outline of the method. A file in a shareable directory is chosen to hold the workstation log. (Probably, this will be in the same directory with the database files themselves.) When the first workstation logs in, this file doesn't yet exist, and the <WSNrLogIn> routine creates the file and writes <MaxNrOfWorkStations> bytes of zeros to it. It then allocates the first workstation number by writing a one to the first byte of the file. When another workstation logs in, it finds the file and takes the next available workstation number, then marks the workstation number used with a one in the workstation log. The workstation number is assigned directly to B-Tree Filer's <IsamWSNr> variable. These operations are done with appropriate file locking and retry to avoid conflicts.

When a workstation is ready to exit the application, it must call the corresponding routine, <WSNrLogOut>, to mark its workstation number as free. These routines are described fully in Section 6.H.

G. The Network Interface

The source code for all network interfaces is found in the file ISNETSUP.INC. The active interfaces are determined at compile time by conditional compilation defines found in BTDEFINE.INC and the selected interface is determined at run time by a parameter passed to <BTInitIsam>. To add a new network interface, follow these steps:

1. Choose an unused name for the new interface, for example, "MyNet".
2. Add the corresponding define in BTDEFINE.INC, e.g., {\$DEFINE MyNet}.
3. Add the new network name to the enumerated type <NetSupportType>, found in FILER.PAS.
4. Near the top of FILER.PAS, add the following sequence, which is used to validate the defines supplied by BTDEFINE.INC:

```
($IFDEF MyNet)
($DEFINE AnyNet)
($ENDIF)
```
5. Near the bottom of FILER.PAS, add the network to the set of accepted networks, as follows:

```
($IFDEF MyNet)
+ [MyNet]
($ENDIF)
```
6. If the network uses Filer's interrupt \$24 service routine, add the following to the beginning of ISNETSUP.INC:

```
($IFDEF MyNet)
($DEFINE IsamINT24HandlerRequired)
($ENDIF)
```
7. At the end of ISNETSUP.INC modify the function <IsamInitNet>. At the end of the Case statement in that routine, add the following new label:

```
($IFDEF MyNet)
MyNet : IsamInitNet := MyNetInitNet;
($ENDIF)
```
8. Edit ISNETSUP.INC to add the network interface routines. Surround these routines with {\$IFDEF MyNet} and {\$ENDIF}. See the existing routines for examples. The routines must be global and compiled with the far model.

Every network interface must provide the following routines (named here for the example network MyNet):

```
function MyNetLockRecord(Start, Len : LongInt; Handle : Word) : Boolean;
```

This function must lock the bytes from <Start> to <Start>+<Len>-1 inclusive in the file with the handle number <Handle>. If successful, the routine must return True, otherwise False.

```
function MyNetUnlockRecord(Start, Len : LongInt; Handle : Word) : Boolean;
```

This function must unlock the bytes from <Start> to <Start>+<Len>-1 inclusive in the file with the handle number <Handle>. If successful, the routine must return True, otherwise False.

```
function MyNetInitNet : Boolean;
```

This function must assign values to <IsamWSNr> and <IsamNrOfWS>, both of which are global Word variables declared by the FILER unit. <IsamNrOfWS> must have the same value on all machines in the network and lie in the range from 1 to <MaxNrOfWorkStations>, inclusive. <IsamWSNr> must be different for every workstation and must lie in the range of 1 to <IsamNrOfWS>. The function must also assign the addresses of MyNetLockRecord, MyNetUnlockRecord, and MyNetExitNet to the global pointer variables <FuncIsamLockRecord>, <FuncIsamUnlockRecord>, and <FuncIsamExitNet>, respectively. It can install an interrupt \$24 service routine by calling <IsamInstallInt24Handler> if needed. The function returns True if successful, False otherwise. An example is provided below.

```
function MyNetExitNet : Boolean;
```

This function must reverse all actions performed by MyNetInitNet. This is especially important for deinstalling an interrupt \$24 service routine, if any. The function returns True if successful, False otherwise.

Here's an example for MyNetInitNet:

```

function MyNetInitNet : Boolean;
var
  WSNr : Word;
begin
  MyNetInitNet := False;
  {Get network-specific unique workstation number}
  WSNr := GetStationNumber;
  {Validate the station number}
  if (WSNr < 1) or (WSNr > MaxNrOfWorkstations) then
    Exit;
  {Store global workstation number and total number of workstations}
  IsamWSNr := WSNr;
  IsamNrOfWs := MaxNrOfWorkstations;
  {Initialize the network if needed}
  {Install interrupt $24 handler if needed}
  IsamInstallInt24Handler;
  {Initialize function pointers}
  FuncIsamLockRecord := @MyNetLockRecord;
  FuncIsamUnLockRecord := @MyNetUnLockRecord;
  FuncIsamExitNet := @MyNetExitNet;
  {Return True for success}
  MyNetInitNet := True;
end;

```

As shown, this function must perform any required initialization for the network. For example, with Novell we must disable error reporting to the screen. With MsNet, we must install a critical error (int 24h) handler to detect attempts to access locked file regions.

If your network requires a critical error handler, you'll probably want to use the existing one in ISNETSUP.INC by calling IsamInstallInt24Handler. If you need to write your own handler, note that it must set the existing Boolean variable <IsamLockError> to True when a lock violation is detected, and return to DOS with the "fail" code set (AL=3).

Although Turbo Pascal 4.0 and later supply their own critical error handlers, these are not adequate for B-Tree Filer's needs, since it must not only set a global flag when an error occurs, but it also must call DOS's "get extended error" service to correctly classify an error as a locking violation.

3. Principles of Operation

This chapter offers some theoretical underpinning for B-Tree Filer, defining the terminology and describing how the fundamental B-tree operations work. If you've already had a course covering B-trees, or if you're not really interested in the internals, you may want to skip directly to Chapter 4.

Software developers use the term "database" so often that its meaning in any particular context is often hazy. In this manual, we use the term database to refer loosely to a collection of data stored for an application, without really specifying what form that collection takes. We will formally define the term "fileblock" to mean the combination of a data file, an index file containing keys, and other information used to manage access to these files. With this terminology in mind, we can proceed with the general topic of data and keys.

A. Data and Keys

There are two basic methods of searching for particular information in a database:

- sequential data access
- indexed sequential data access, also known as random access

Searching by sequential access entails reading the data file from beginning to end while comparing every data record to the desired one. This is slow and inefficient with large data sets. To access the data directly one must know the position of the desired data record in the sequential data file. Given the position, one data access will read the correct record.

The information needed to find the data record is available with the help of the index key information stored by B-Tree Filer. Data records and keys are stored in two independent files, the data file and the index file.

The data file consists of fixed size data records stored one after another. Every data record corresponds to a number, the data record reference, defining the position of the data record in the data file. For example, reference 0 refers to the very first record in the data file (reserved by B-Tree Filer for internal use), and reference 1 is the first user record in the file. When a record is entered into the data file with the B-Tree Filer routine <BAddRec>, the data record reference is returned, and can then be added to the index file with its corresponding key. The index file contains the keys and their corresponding record reference values. (The terms "data record reference" and "record number" are used interchangeably in the documentation.)

To find a certain data record, one searches in the index file for the key by using one of the B-Tree Filer routines, e.g., <BFindKey>. If the key is found, the data record can be read using another B-Tree Filer procedure, <BGetRec>, using the reference number returned by the search procedure. If the key is not found, the B-tree search method guarantees that the data record doesn't exist.

B. B-tree Organization

Trees are one of the most-studied data structures in computer science, and the B-tree is among the most popular trees. Many people think that "B-tree" means binary tree, but that's not so. The "B" in B-tree stands for "Bayer", who is one of the inventors of this data structure. The B-tree is a much richer data structure than the binary tree, and one that was designed specifically to optimize searching for records in large databases. The remainder of this section defines the terminology of B-trees in general, then describes how B-trees are applied to database management. The following section covers the basic B-tree algorithms for searching the tree, and for adding and removing elements.

B-tree Terminology

The term "tree structure" comes from the similarity of this general data structure to an upside-down tree. A "node" in a B-tree represents a collection of one or more index keys. The "branches" connecting the nodes denote a search order to be used while finding a particular key value. The branches of the tree point downward, starting at the "root", continuing until nodes with no outgoing branches are encountered. These nodes are called "leaves". (See Figure 1, below.)

The height of the tree is defined by the maximum number of branches traversed while going from the root to any leaf. The root is at level 1, its direct branches at level 2, with its direct descendents at level 3, etc. All leaf nodes, i.e., all nodes that don't have any descendents, end at the same level in the B-tree structure.

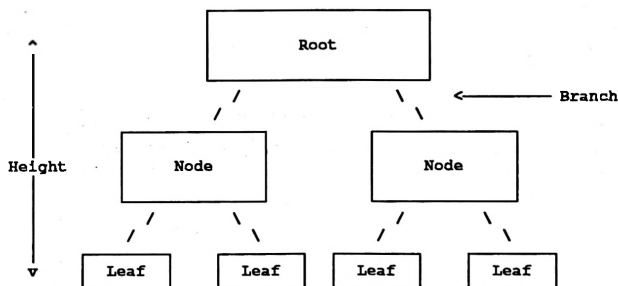


Figure 1: B-tree of Order 2 and Height 3

The number of branches leaving any given node is called the "degree" of the node, with the largest degree giving the degree of the tree. For example, a binary tree has a degree of two, since each node has two branches. Multi-branched trees, e.g., the B-tree, have a degree greater than two. In B-Tree Filer, the degree of the tree is set implicitly with the constant <PageSize> to the value <PageSize>+1.

The "order" of a tree is defined as the minimum number of branches on any node. Every node in a B-tree, not including the root, must be at least half filled; therefore every node, with the exception of the root, must have at least $\lceil \text{PageSize} / 2 \rceil + 1$ branches. This leads to efficient memory usage.

A "balanced" tree is one in which the levels of the leaf nodes vary by at most one. The B-tree optimizes this even further by requiring all leaf nodes to be at the same level. This implies that no matter which path from the root is taken, the number of steps to the leaf node is always the same.

The construction of a B-tree, in contrast to a binary tree, uses more memory but guarantees fewer accesses to the disk and therefore faster data access times.

B-trees for Databases

The B-tree is organized as a "search tree", i.e., the keys are stored in a certain order in the tree. The keys used to access the records in the data file are stored in the nodes of the tree.

The index entries are stored in increasing order within each node. The first index entry contains the smallest key of the node; the last index entry contains the largest key of the node.

The index entries are defined as records containing the following fields:

- the associated key
- the data record reference
- forward node reference

The node reference (forward pointer) points to the node containing keys larger than the one stored in the current index entry. (See Figure 2, below.)

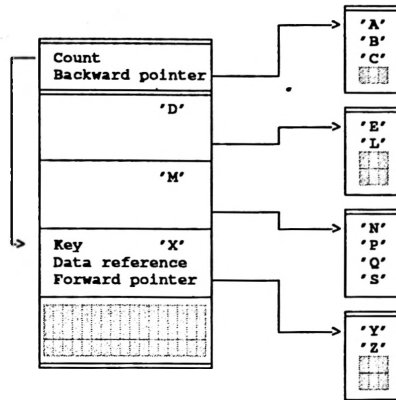


Figure 2: Internal Structure of a B-tree

The node is itself declared as a record and contains:

- an integer holding the actual number of index entries stored on this node
- a backward node reference (backward pointer) pointing to the node that contains keys smaller than any keys in this node
- an array of the index entries themselves

The B-tree can be traversed from node to node through the use of the backwards pointer, which is used to find smaller keys, and the forward pointer, used to find larger keys.

The smallest key of the B-tree is the first element within the leaf node at the leftmost edge of the tree (which has no backwards pointer, since there are no keys smaller), and the largest key is the last element within the rightmost leaf node (with no forward pointer, since there are no larger keys). The search path (starting from the root) to find either the smallest or largest key is of equal length since all leaf nodes of a B-tree must lie at the same level.

Every node contains at least $\lceil \text{PageSize}/2 \rceil$ and at most PageSize index entries. The root is the sole exception in that it may contain less than $\lceil \text{PageSize}/2 \rceil$ index entries.

Choosing a value for PageSize involves tradeoffs. The search for keys in larger pages requires fewer accesses to the disk, and is therefore faster than having smaller pages. The search within a page is considerably faster than loading a page from disk. However, a larger PageSize increases the memory requirements of B-Tree Filer,

and it also increases the time to read a page. The default value of 62 for <PageSize> has been selected based on thorough experimentation, and guarantees optimal access times and appropriate memory usage.

B-Tree Filer also defines the maximum height of the B-tree with the constant <MaxHeight>. With the default value of 8, the B-tree will not fill until more than 10^{10} keys are added to it. Since 2^{32} keys is less than 10^{10} , there is no danger of the B-tree overflowing!

Keys in the Index File

As already described, the keys in the index file provide the connection to the data records in the data file. Every time a data record is entered, at least one corresponding key must be added so that the data record may be found through an indexed access.

Since the index and data files are stored separately, either file may be manipulated independently. If a key is deleted, the corresponding data record is not automatically deleted. In this case, the data record cannot be found by an indexed access. An indexed access is also not possible if a data record is entered in the data file without simultaneously adding the key entry to the index file. Parallel manipulation of the index and data files is therefore essential for proper functioning of B-Tree Filer.

All index keys are of type String (<IsamKeyString>). If a number, e.g., an invoice number, is to be used as a key, it must first be converted to a string. (Routines to perform such conversions may be found in the NUMKEYS unit, discussed in Section 6.G.) The value of each element in the string is ASCII based. Care must be used when building the string (e.g., usage of capital and lower case letters) to insure that keys may be sensibly compared with one another.

There are two types of keys, distinguished as follows:

- primary keys: keys that are unique within the database
- secondary keys: keys that may be entered multiple times

Primary keys may not be duplicated within the index file (this requirement is enforced by B-Tree Filer's indexing routines). Customer numbers, invoice numbers, etc., may be used as primary keys.

Secondary keys are used for non-unique items: names, addresses, weights, etc. B-Tree Filer keeps these possibly duplicated secondary keys separate internally by combining the unique data record references with each key. (The process of combination is done logically, and requires no additional space for key storage.)

The B-tree does not require the existence of a primary index since internally all keys are made unique anyway.

The maximum length of a key is determined by <MaxKeyLen>, which may lie between 1 and 255 (default value is 30). The length and type of each key are determined by the values of the fields <KeyL> and <AllowDupK>, specified within a variable of B-Tree Filer type <IsamIndDescr>.

Every key corresponds to exactly one data record, though many keys may point to a data record. In an address data base, the name, the address, as well as any other item may be made into a key. The number of keys that may point to a data record is determined by the parameter <NumberOfKeys> that is passed to the procedure <BTCreatFileBlock>. In B-Tree Filer, no matter how many keys are used to manage a given database, all those keys are stored within a single index file.

C. Managing Keys

As we shall see, adding or deleting a single key can require a major reorganization of the B-tree in order for it to retain its required properties. In this section, we describe how B-Tree Filer transparently manages the B-tree.

Searching for a Key

The search for a key in the B-tree always begins at the root node. The root, as well as every other node that must be loaded, is searched using a binary search. The number of index entries currently stored on the node is halved and the middle index entry obtained. Its key entry is then compared to the desired key. If they are identical, the search is ended successfully.

The search must proceed further if the key entry does not match the desired key. Since the keys on a node are always stored in increasing order, the comparison of the present key with the desired key provides the direction for further search. If the compared key entry is smaller than the desired key, the search proceeds to the right. If it is larger than the desired key, the search goes left. If the key is found anywhere on the present node, the search is ended.

Otherwise, the backwards and forwards pointers can be used to locate another node. The forward pointer of the last index entry whose key was smaller than the desired key leads to a new node with keys larger than the last entry and smaller than the next entry of the original node. The search then continues on the new node, which may already be in memory or may require a disk access to obtain. Only when the search proceeds to a node where the smallest index entry is larger than the search key is the backwards pointer used. The search then continues on this node, where the keys are all smaller than the one just searched. The search ends successfully when the corresponding key is found or in failure when a leaf node of the tree is reached without a match.

The B-tree can also access the records of a database in sorted sequential order. From a given starting point, the forward pointer of the current index entry leads to the next key in sorted order. When a leaf node is reached, the algorithm backs up to the previous node level and continues with the next larger index entry. This is equivalent to a depth-first traversal of the B-tree. To denote a particular location within the B-tree, the term "sequential pointer" is used. In reality the sequential pointer is an array of records, each record containing one number that defines the node and another that indicates the active element within the node. The sequential pointer can be thought of as a path leading from the root to the node containing the current entry. Certain B-Tree Filer routines initialize the sequential pointer to a known value. Other routines overwrite the pointer with a temporary value. Descriptions of the routines will point out their effect on the sequential pointer.

Adding a Key

If a new key is to be added to the index file, the B-tree is first searched using the just described method to assure that the key doesn't already exist. If the key does exist, an error is generated. (Remember that duplicate keys are made unique by incorporating their data record reference numbers.) If the key does not exist, the search ends at a leaf node with negative results. The new key is added at this location. If there is still room for another key on that leaf node, the addition is performed by adding the new key in sorted position within the node's index array. However, if the leaf node is full--there are already $\langle \text{PageSize} \rangle$ keys stored there--the B-tree must be modified to make room for the new key. (See Figure 3, below.)

The required space is created by splitting this node in two. The $\langle \text{PageSize} \rangle / 2$ smaller index entries remain on the old node, while the $\langle \text{PageSize} \rangle / 2$ larger index entries are entered on a new node. The middle index entry, which is smaller than the index entries of the new node, is used to point to this new node, by entering it in a node of a previous level of the tree.

If the node that this middle index entry is to be entered into is also full, another node split occurs. By recursively applying this method there may be node "halvings" all the way to the root level. Even the root may then be split as described. In this case, the middle index entry from the final split will become the new root and the tree grows by one level. This is the only case where the height of the tree grows. This method of adding keys preserves the distinctive characteristic of a B-tree, that all leaf nodes must lie at the same level.

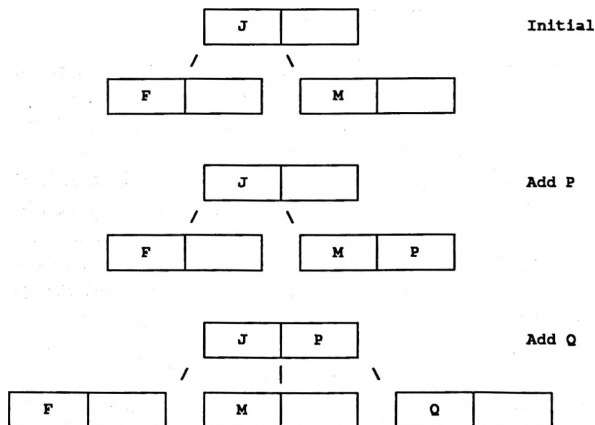


Figure 3: Adding Keys to a B-tree

In implementing node splitting, B-Tree Filer offers a further enhancement of basic B-tree theory.

Before a node is split, an attempt is made to perform a "page balance", which will empty some of the node. Here, the number of index entries on a neighboring node is checked. If there are fewer than <PageSize> entries on the neighboring node, index entries from the overflowing node are moved over to the neighboring node. A node split is then no longer necessary.

This method is better in that it allows for filling all nodes with keys, and therefore it conserves memory. This leads to the following benefits:

- the search time is decreased
- the size of the index file is decreased
- more keys can be kept in memory

When a continuous, sequential key (a customer number, for example) is used, the index file size may be reduced by 30% to 50%.

Deleting a Key

If a key is to be deleted, the location of the key in the tree must first be determined. This is accomplished in the manner previously described.

As with addition, it is easy to delete a key found on a leaf node. This is done by simply removing the entry from the node's index array.

If by the deletion the node now has less than <PageSize>/2 index entries ("underflow"), either of two actions may be taken. Index entries can be obtained from a neighbor node ("page balancing"), as described in the addition of keys. Or in the case where the neighbor would also suffer from underflow, the two pages will be made into one ("node merging", exactly the opposite of node splitting). The parent of the two pages, now merged into one, also shrinks by one index entry. If it also should suffer from underflow, and page balancing is not possible, it also would combine into one node with a neighbor. Under certain conditions this will occur all the way to the root. If this should happen to remove the single index entry that made up the root, then the tree would shrink by one level. This is the only way that a B-tree can lose height.

It is mentioned again that the index and data files are stored separately. If an index entry is deleted without deleting the corresponding data record, the data record remains as a "data orphan" in the data file, i.e. it can no longer be found in the data file through an indexed reference.

If a key is to be deleted from a location other than a leaf node in the tree, the forward pointer that points to the node with the larger keys would also be lost by the deletion of the key, and the structure of the tree would be corrupted. In order to prevent this, the next largest key in the tree is obtained and moved to the location that the deleted key occupied. Because of the key ordering this will always be located on a leaf node. In this manner the forward pointer of the deleted key remains valid. After the new key has been copied to the location occupied by the deleted key, it must also be removed from its previously occupied position. This is done with the same scheme by which a key is deleted from a leaf node. In case the node from which the deleted key was removed now has less than <PageSize>/2 entries, it is corrected by either node balancing or node merging.

4. Using B-Tree Filer

Compared to Chapter 3's dense conceptual descriptions, this chapter will offer a welcome breeze of practicality. Here, we show how to put B-Tree Filer to use in your applications. The programming examples of sections C and F will provide a quick start for those who learn best by example.

A. Fileblocks

As already mentioned, a "fileblock" is the combination of a data file, an index file, and other information kept in memory and on disk to describe the state of the data and index files. For single user databases, the "other information" is kept in memory and in the first record of the data file, which is always reserved for B-Tree Filer's use. For databases opened in save mode and for multi-user databases, the fileblock incorporates a third file, called the dialog file. In save mode, this file temporarily holds records or keys until they are successfully written to the actual data or index file. In multi-user applications, the dialog file holds additional information required to manage the interactions among workstations.

B-Tree Filer can manage any number of fileblocks, constrained only by the number of open files allowed by the operating system. There is only one index file for every data file, independent of the number of keys defined. Therefore, if five single user fileblocks are to be opened at one time, the operating system must be able to keep ten different files open. If three multi-user fileblocks are to be opened, nine DOS files will be required.

(MS-DOS controls the maximum files per process with the FILES= entry in the CONFIG.SYS file. Since the first five files are always used by DOS, be sure to account for those in the total request. Without extensions, DOS allows at most 20 simultaneously open files in a given program. B-Tree Filer offers the <ExtendHandles> procedure to increase the number of files under DOS 3.3 or later. See Section 6.H for more information.)

B-Tree Filer databases are referenced through a pointer to a fileblock. This minimizes static data allocation by using the heap to hold the fileblock information that is kept in memory. An application declares a pointer variable of type <IsamFileBlockPtr> for each fileblock, which consumes only four bytes in the data segment.

When a fileblock is opened, several boolean parameters are used to specify properties of the fileblock. These parameters are described fully in Chapter 5's

description of <BOpenFileBlock>, but we'll mention two of them here. The first one, <Net>, specifies whether the fileblock is stored on a file server and is to be used by more than one workstation simultaneously. The second one, <Save>, determines whether the fileblock is opened in "save mode", a special mode that B-Tree Filer provides to increase data integrity.

In save mode, B-Tree Filer uses the dialog file to keep a backup of each modified portion of a fileblock while changes are being made. In that way, if an error occurs during the operation, changes can be undone and the fileblock restored to a known condition. If the operation is successful, all changes are flushed to disk and the backup information in the dialog file is removed. Even though the dialog file holds backup information for only a single atomic operation (adding a key or adding a record, for example), save mode can increase data integrity enormously. Reliability comes at the expense of performance, however, so save mode is not always desirable. See Section 4.G for a discussion of the pros and cons.

B. Program Organization

To access B-Tree Filer's routines, Use the FILER unit in your main program and in any other units where needed. The program must be organized with the following sequence of actions.

1. Initialize the FILER unit and create a page buffer by calling <BTInitIsam>.
2. Optionally create one or more fileblocks by calling <BTCreateFileBlock>.
3. Open one or more fileblocks by calling <BTOpenFileBlock>.
4. Use the B-Tree Filer procedures and functions on the opened fileblocks.
5. Close opened fileblocks with <BTCloseFileBlock>.
6. Dispose of the page buffer and shut down the FILER unit by calling <BTExitIsam>.

This sequence may be repeated as often as desired. In addition, files may be opened and closed as desired between steps two and five, inclusive.

Another important program action should be considered at the highest level, and that is error handling. B-Tree Filer exports the boolean variable <IsamOK>, which is set True for success, and False when an error has occurred. An integer variable, <IsamError>, is then available for classification of the error. Almost every call to a B-Tree Filer routine should be followed by a check of <IsamOK>.

The value of the parameter <ExpectedNet> passed to <BTInitIsam> determines whether B-Tree Filer is dealing with a real network or is emulating a network on a single PC. By passing <NoNet> in this parameter when the FILER unit has been compiled with any network define other than NoNet (in BTDEFINE.INC), you can develop a network application on a PC not currently connected to the network, or you can develop an application that runs in single user mode today but is ready for a multi-user upgrade in the future. Of course, you can't be sure your locking logic is correct until the application is running in the real environment.

Many single user applications must eventually be ported to a network environment. To simplify this task, all B-Tree Filer procedures and function calls are available in the single user as well as the network version. If you call a routine such as locking a fileblock that has no relevant action in the single user version, it does nothing at all but return a successful status code.

C. Single User Example

The following programming examples illustrate operations on B-Tree Filer fileblocks. Each major action--defining the record structure, opening the database, adding and deleting records and keys, and searching for information--is shown. You'll note that many of the routines are simply shells around Filer's lower level routines, adding error handling and specific information about the particular application. This approach is recommended for all applications based on B-Tree Filer.

The following code examples can be found in the file EXAMPLE.PAS. (EXAMPLE.PAS doesn't actually do anything, but you may find it useful for cutting and pasting.)

Data Definition

One must first define a record type to hold the information stored by the database. It could be like the following:

```
type
  PersonDef =
    record
      Del : LongInt;
      FirstName : String[20];
      LastName : String[25];
      Street : String[30];
      City : String[30];
      State : String[2];
      ZipCode : String[9];
      Telephone : String[15];
      Age : Integer;
    end;
```

The first field, , of type LongInt, must be initialized to zero by the application. We recommend that all data records begin with such a field. B-Tree Filer overwrites these first four bytes when it deletes the record, in order to maintain a linked list of reusable space within the data file. Many of B-Tree Filer's facilities, particularly those for rebuilding and reorganizing fileblocks, determine whether a given record is valid by checking whether the first four bytes of the record are zero or not. (Deleted records always have a non-zero value, valid records should have zero.) These facilities will not work if a field such as isn't defined.

The other fields here are given for purposes of example. B-Tree Filer's only restriction is that the record be 21 bytes or larger in size. This restriction is imposed by the fact that the first record of any data file holds information describing the database as a whole, allowing B-Tree Filer to open it without additional files.

To access the corresponding fileblock, a variable of type <IsamFileBlockPtr> must be declared. Typically, this variable will be global to an entire program.

```
var
  PF : IsamFileBlockPtr;           (Symbolic access to the database)
```

Although the following routines access this global variable directly, they could just as well pass an <IsamFileBlockPtr> as a parameter. When an application is managing multiple fileblocks, it's recommended to pass the fileblock pointers as parameters. See the BONUS diskette for an example of using more than one fileblock in a single program.

Allocating Page Buffers

B-Tree Filer uses page buffers to hold as many B-tree nodes in memory as will fit, thereby optimizing search times. Its algorithm requires that at least <MaxHeight> nodes be held in memory at once. The <BTInitIsam> routine is used to allocate these page buffers. It allows you to reserve heap space for other uses, then allocates the remaining space for buffers.

```
procedure AllocatePageBuffer(HeapToRemain : LongInt);
var
  NumberOfPages : LongInt;
begin
  NumberOfPages := BTInitIsam(NoNet, HeapToRemain, 0);
  if not IsamOK then begin
    (Insufficient memory)
    Halt;
  end;
end;
```

In this case <BTInitIsam> returns an error only if it was unable to allocate at least <MaxHeight> pages. If <IsamOK> is False, the returned value contains the number of pages that it could have allocated, but didn't. The typical response to an insufficient memory error is simply to write an error message and halt. Note that <BTInitIsam> is also capable of using EMS memory for the page buffers; see Chapter 5 for more details.

Creating a Fileblock

It is to your advantage to declare global constants describing the length of each key in a fileblock. These will be used in more than one place and when the inevitable day comes when you want to adjust the key definitions, you'll appreciate having the constants.

```

const
  Key1Len = 30;           {First and last name}
  Key2Len = 5;            {ZipCode}
  Key3Len = 15;           {Telephone}

function CreateFile : Boolean;
var
  IID : IsamIndDescr;
begin
  IID[1].KeyL := Key1Len; IID[1].AllowDupK := False;
  IID[2].KeyL := Key2Len; IID[2].AllowDupK := True;
  IID[3].KeyL := Key3Len; IID[3].AllowDupK := True;
  BTreeCreateFileBlock('TEST', SizeOf(PersonDef), 3, IID);
  CreateFile := IsamOK;
end;

```

CreateFile will create the necessary data and index files for a new fileblock. In our example, <BTreeCreateFileBlock> creates the data file 'TEST.DAT' and the index file 'TEST.IX'. The Pascal function SizeOf provides the length of the data record, i.e., the record size.

The parameter that describes the keys is defined in the variable IID of type <IsamIndDescr>. IID must be initialized before calling <BTreeCreateFileBlock>. In the example, there are three keys for every data record. (B-Tree Filer allows up to 100 keys for every record, with each key up to 30 characters long. These values may be adjusted by changing constants in FILER.CFG. See Section 5.A for more details.) The <IsamIndDescr> describes each key, providing the maximum length of each string and whether the key is a primary (unique) or secondary (possibly duplicate) key. Keys may be created from any data desired; they are not restricted to data from any one field. In our example, the primary key will be the concatenation of the last and first name. The zip code and telephone number will be entered as secondary keys. Note that B-Tree Filer does not require a primary key for each fileblock.

After a fileblock has been created, it must still be opened in order to use it.

Opening a Fileblock

```

function OpenFile : Boolean;
begin
  BTreeOpenFileBlock(PF, 'TEST', False, False, False, False);
  if not IsamOK then begin
    OpenFile := False;
    {Error reporting code that examines
     <IsamError> can go here. Corrective action may
     be taken, for example by reconstructing a defective
     index file as described in Section 6.D.}
    Exit;
  end else
    OpenFile := True;
end;

```


OpenFile shows how a function to open an existing fileblock might look. It contains a call to the Filer procedure <BTOpenFileBlock> that opens the fileblock in normal mode for single user access. The four boolean parameters passed to <BTOpenFileBlock> specify alternate open modes and are described in Chapter 5. By using the global variables <IsamOK> and <IsamError>, it is possible to detect errors that occur while opening the files. Notice that no information is needed about the record length or the keys to open an existing fileblock. B-Tree Filer reads this information from the data file and keeps the information in the corresponding fileblock in memory.

Closing a Fileblock

```
function CloseFile : Boolean;
begin
  BTCloseFileBlock(PF);
  if not IsamOK then begin
    CloseFile := False;
    {Error handling}
    Exit;
  end else
    CloseFile := True;
end;
```

It is essential that every open fileblock be closed before an application halts. Otherwise buffered index information may not be written to disk and an error will occur during the next use of the fileblock. The function CloseFile can handle this task. It closes the index and data files by calling <BTCloseFileBlock>. Error handling may take place here as well, although at this point there are few corrective actions to take, short of rebuilding the fileblock.

Data and Index File Operations

It is advantageous to write a single function that returns the keys with which the data file may be accessed. This ensures that keys associated with a given data record will be consistent throughout an entire application. CreateKey is an example of such a function, creating any of the three different key strings from the record fields.

```
($F+) {Routine should be global}
function CreateKey(var P; KeyNr : Integer) : IsamKeyStr;
begin
  with PersonDef(P) do
    case KeyNr of
      1 : CreateKey := StUppcase(Pad(LastName, 20)+Pad(FirstName, 10));
      2 : CreateKey := Copy(ZipCode, 1, 5);
      3 : CreateKey := Copy(Telephone, 1, 15);
    else
      CreateKey := '';
    end;
  end;
end;
```

We've made this function a little more complicated than it needs to be now, so that it will also work with other B-Tree Filer units to be described later. Units such as REORG and REBUILD call a routine like CreateKey indirectly when they are rebuilding an index file. The routine must meet specific requirements in order for the indirect call to work correctly.

CreateKey is declared far and global by using the {\$F+} directive (or the new Far keyword in Turbo Pascal 6). The routine's first parameter is an untyped Var parameter and the second parameter is the KeyNr of type integer. The Var parameter passes the data record into the routine. Because it's untyped it will work regardless of the record type; however, you'll note that the routine must typecast the parameter into the actual data type in order to refer to the fields of the record.

StUppcase is a function that converts the argument string to upper case. Pad is a function that either truncates a string to the specified length or pads it with blanks to reach that length. It's important that the CreateKey routine never return a key string longer than the allowed length for a given index; otherwise routines such as <BTAddKey> will generate an error. It's not essential for CreateKey to pad all key strings to the same length. However, when two fields are concatenated to create a key string, as with first and last names here, it is important to pad the first field to its maximum length before adding the second field. Otherwise, the index sorting sequence will not be correct.

Now that files exist and keys may be created, one needs routines to add, delete, change, and search for data.

Adding Data to the Fileblock

Inserting a new record requires two steps. <BTAddRec> is called first to add the record to the data file. This routine returns a <RefNr> (data record reference) to describe the record's location within the file. The second step requires adding every key to the index file with <BTAddKey>. The following example should clarify the situation:

```

procedure UndoAdd(P : PersonDef; RefNr : LongInt; LastKey : Integer);
var
  KeyNr : Integer;
  Key : IsamKeyStr;
begin
  for KeyNr := 1 to LastKey do begin
    Key := CreateKey(P, KeyNr);
    BDeleteKey(PF, KeyNr, RefNr, Key);
    if not IsamOK then
      (Abort: too many errors)
      Halt;
    end;
  end;
end;
```

```

function AddRecord(P : PersonDef) : Boolean;
var
  KeyNr : Integer;
  RefNr : LongInt;
  Key : IsamKeyStr;
begin
  AddRecord := False;
  BAddRec(PF, RefNr, P);
  if not IsamOK then begin
    {Error handling}
    Exit;
  end;
  for KeyNr := 1 to BNumOfKeys(PF) do begin
    Key := CreateKey(P, KeyNr);
    BAddKey(PF, KeyNr, RefNr, Key);
    if not IsamOK then begin
      {Remove keys added so far}
      UndoAdd(P, RefNr, KeyNr-1);
      {Remove the new record}
      BDeleteRec(PF, RefNr);
      {Error handling}
      Exit;
    end;
  end;
  AddRecord := True;
end;

```

After the new data record has been successfully entered into the file with <BAddRec>, the keys are created one by one within a for loop, and are added to the index file using <BAddKey>. The variable RefNr contains the location to which the new data record was written. If errors are encountered during the key operations (e.g. <IsamError> = 10230, duplicate key), the function UndoAdd attempts to delete all keys added so far, so that AddRecord can return gracefully by not corrupting the fileblock. However, if more errors are encountered during the deletion, attempting further recovery will probably cause more problems than it solves, so the program should be aborted. AddRecord returns True if all operations are successful.

Removing Data from a Fileblock

The function DeleteRecord is analogous to AddRecord.

```

procedure UndoDel(P : PersonDef; RefNr : LongInt; LastKey : Integer);
var
  KeyNr : Integer;
  Key : IsamKeyStr;
begin
  for KeyNr := 1 to LastKey do begin
    Key := CreateKey(P, KeyNr);
    BAddKey(PF, KeyNr, RefNr, Key);
    if not IsamOK then
      Halt;
  end;
end;

```

```

function DeleteRecord(P : PersonDef; RefNr : LongInt) : Boolean;
var
  KeyNr : Integer;
  Key : IsamKeyStr;
begin
  DeleteRecord := False;
  {Assure record not already deleted}
  if P.Del <> 0 then
    Exit;
  for KeyNr := 1 to BTNrOfKeys(PF) do begin
    Key := CreateKey(P, KeyNr);
    BTDeleteKey(PF, KeyNr, RefNr, Key);
    if not IsamOK then begin
      {Add back keys that have been deleted so far}
      UndoDel(P, RefNr, KeyNr-1);
      {Error handling}
      Exit;
    end;
  end;
  BTDeleteRec(PF, RefNr);
  if IsamOK then
    DeleteRecord := True;
end;

```

To prevent disastrous consequences that will occur by deleting an already deleted record, the P.Del field is examined to confirm that the data record has not yet been deleted. Then one proceeds in just the opposite manner compared to adding a record. First the keys are deleted, and then the data record. It's debatable whether to attempt to undo a deletion in case of an error. We've shown UndoDel here, but you may decide it makes sense to continue deleting keys even if one <DeleteKey> fails.

Modifying Records

When an existing record is to be modified, not only must the data file be updated, but also the old keys must be deleted and new keys added. This makes for a fairly complicated procedure, as shown in ModifyRecord.

```

function CheckRecord(P, POld : PersonDef) : Boolean;
begin
  {Verify that: new record has valid keys,
   new record differs from old}
  CheckRecord := True;
end;

function ModifyRecord(P, POld : PersonDef; RefNr : LongInt) : Boolean;
var
  KeyNr : Integer;
begin
  ModifyRecord := False;
  if not CheckRecord(P, POld) then
    Exit;
  for KeyNr := 1 to BTNrOfKeys(PF) do begin

```

```

(Update modified keys)
if CreateKey(P, KeyNr) <> CreateKey(POld, KeyNr) then begin
  BDeleteKey(PF, KeyNr, RefNr, CreateKey(POld, KeyNr));
  if not IsamOK then
    if IsamError = 10220 then
      (Key already deleted, ignore the error)
    else begin
      UndoAdd(P, RefNr, KeyNr-1);
      UndoDel(POld, RefNr, KeyNr);
      Exit;
    end;
  BTAddKey(PF, KeyNr, RefNr, CreateKey(P, KeyNr));
  if not IsamOK then begin
    UndoAdd(P, RefNr, KeyNr-1);
    UndoDel(POld, RefNr, KeyNr);
    Exit;
  end;
end;
end;
BTPutRec(PF, RefNr, P, False);
if not IsamOK then begin
  UndoAdd(P, RefNr, BTNrOfKeys(PF));
  UndoDel(POld, RefNr, BTNrOfKeys(PF));
  Exit;
end;
ModifyRecord := True;
end;

```

The CheckRecord routine would verify that the new record, P, generates valid keys (e.g., its primary keys must not be empty). CheckRecord would also check that the new and old records weren't identical, since that would lead to a lot of wasted effort.

ModifyRecord removes each changed key from the index file and adds the new key. If an error occurs, it calls UndoAdd and UndoDel to recover as gracefully as possible from the error. Then it calls <BTPutRec> to overwrite the old data record with the new.

Next or Previous Data Record

NextPrevRecord shows how a function can be implemented to scan through a data file in index order. The "sequential access pointer" must be positioned over a known key before NextPrevRecord can be called for the first time. This can be done by calling any of the functions <BTFindKey>, <BTSearchKey>, <BTFindKeyAndRef>, <BTSearchKeyandRef>, or <BTClearKey>.

```

function NextPrevRecord(var P : PersonDef;
                        var RefNr : LongInt;
                        KeyNr : Integer;
                        var Key : IsamKeyStr;
                        Next : Boolean) : Boolean;
begin
  NextPrevRecord := False;
  if Next then begin

```

```

    BTNextKey(PF, KeyNr, RefNr, Key);
    if not IsamOK and (IsamError = 10250) then
        {There was no next key. Move to first key in the file}
        NextKey(PF, KeyNr, RefNr, Key);
    end else begin
        BTPrevKey(PF, KeyNr, RefNr, Key);
        if not IsamOK and (IsamError = 10260) then
            {There was no previous key. Move to last key in file}
            BTPrevKey(PF, KeyNr, RefNr, Key);
        end;
        if not IsamOK then
            Exit;
        BTGetRec(PF, RefNr, P, False);
        if not IsamOK then begin
            {Error handling}
            Exit;
        end;
        NextPrevRecord := True;
    end;
end;

```

Searching for Data Records

In order to locate a data record by indexed access, the search key string must be specified along with the corresponding index number. The following routine demonstrates an indexed search.

```

function FindRecord(var P : PersonDef;
                    var RefNr : LongInt;
                    KeyNr : Integer;
                    var Key : IsamKeyStr) : Boolean;
begin
    FindRecord := False;
    BTSearchKey(PF, KeyNr, RefNr, Key);
    if not IsamOK then begin
        {Determine why SearchKey failed, for example:
         IsamError = 10210 Neither the key nor any larger was found.}
        Exit;
    end;
    BTGetRec(PF, RefNr, P, False);
    if not IsamOK then begin
        {Error handling}
        Exit;
    end;
    FindRecord := True;
end;

```

The subroutine is aborted if <IsamOK> is False after calling <BTSearchKey>. <SearchKey> returns True even if no exact match was found, as long as another record with a larger key is available. The actual key is returned to FindRecord's caller through FindRecord's Var parameter, Key. If any matching key is found, <BTSearchKey> also returns the associated data record reference number in RefNr. Using this number, the data record is then read by <BTGetRec> and returned in P. The procedure <BTFindKey> should be used instead of <BTSearchKey> if an exact key match is required.

Searching for a record based on a non-indexed field is a little trickier, and can be handled in more than one way. One method ignores index order altogether and simply reads each record from the data file, skipping any records marked as deleted. Another method is to use the <BTNextKey> routine to search the records in sorted order by key. This method is advantageous when a two part search is required: the first part an indexed search on a possibly duplicate key, and the second part a non-indexed search on a different field. Using <BTNextKey> allows the routine to start on the first possible match and to quit as soon as no more matches are possible. ScanForRecord exemplifies this method:

```
function MatchedRecord(P, Q : PersonDef) : Boolean;
begin
  {Return True if P and Q match based on some criteria, for example...}
  MatchedRecord := (StUppcase(P.City) = StUppcase(Q.City));
end;

function ScanForRecord(var P : PersonDef; KeyNr : Integer;
                      var RefNr : LongInt) : Boolean;
var
  Done : Boolean;
  Goal : PersonDef;
  Key : IsamKeyStr;
begin
  ScanForRecord := False;
  Goal := P;
  Done := False;
  repeat
    BTNextKey(PF, KeyNr, RefNr, Key);
    if not IsamOK then
      {Probably reached the largest key}
      Done := True
    else begin
      BTGetRec(PF, RefNr, P, False);
      if not IsamOK then begin
        {Error handling}
        Done := True;
      end else if MatchedRecord(P, Goal) then begin
        {Found a match}
        Done := True;
        ScanForRecord := True;
      end;
    end;
  until Done;
end;
```

This example searches for a record that "matches" the initial record passed in the P parameter. The search begins with the record one beyond that associated with the current sequential pointer, and continues on to the record having the largest key of type KeyNr. The MatchedRecord routine compares the current record to the search goal and, if a match is found, ScanForRecord returns the record number (in RefNr) and its contents (in P).

We have presented the most important operations of B-Tree Filer through these small examples. These routines have been kept simple in order to illustrate the concepts clearly. More robust examples may be found in the NETDEMO example program.

D. Locking Techniques

B-Tree Filer supports six levels of locking, as explained in this section. After each level, we list the routines that activate and deactivate such locks. In every case, it is important for a workstation to minimize the time that it retains a lock.

If you have the single user version of B-Tree Filer, calling the locking routines does nothing.

Level 1: <BTLockAllOpenFileblocks> and <BTUnlockAllOpenFileblocks>

These procedures act upon all network fileblocks that are open at the time of the call. After <BTLockAllOpenFileBlocks> returns successfully, all network fileblocks of the calling workstation are available for its exclusive use. No other workstation may read or write the locked fileblocks. With rare exceptions, a network fileblock must be locked before it can be modified by using <BTAddRec>, <BTAddKey>, <BTDeleteRec>, <BTDeleteKey>, or <BTPutRec>.

No "deadlock" problems can occur with level 1 locking. (A deadlock occurs when two workstations are simultaneously trying to lock the same two files, and one of them locks the first file while the other one locks the second file. Then both could possibly wait forever waiting for the other file to become free.) <BTLockAllOpenFileBlocks> either locks all open network fileblocks, or it locks none of them. The application should check the value of <IsamOK> to determine whether <BTLockAllOpenFileBlocks> was successful.

It is especially critical when using level 1 locking to minimize the amount of time the fileblocks remain locked. One specific situation to avoid is waiting for interactive keyboard input before unlocking again.

Level 2: <BTLockFileBlock> and <BTUnlockFileBlock>

These procedures affect a single network fileblock. After <BTLockFileBlock> returns successfully, the calling workstation has exclusive access to the specified fileblock and may modify it using the procedures mentioned above.

A deadlock may also occur as described under level 1 locks. The calling program must take steps to prevent this situation. Section 4.F below offers methods to avoid deadlocks.

Level 3: <BTLockRec> and <BTUnlockRec>

<BTLockRec> locks a single record of a specific network fileblock. This locking step may be combined with the locking steps 1 and 2. Possible deadlocks must be avoided by the calling program.

Level 4: Not directly implemented by B-Tree Filer

The user may supply routines to use any other locking methods that pertain to a data record (e.g., a "shareable lock" under Novell). B-Tree Filer supports this through the <BTGetRecordInfo> procedure. Given an open fileblock and a record number, <BTGetRecordInfo> returns the handle of the data file that contains the record, as well as the record's file offset and length. This is sufficient for a custom locking routine to access the record.

Level 5: <BTReadLockAllOpenFileBlocks> and <BTUnlockAllOpenFileBlocks>

These procedures act upon all network fileblocks that are open at the time of the call. After <BTReadLockAllOpenFileBlocks> returns successfully, no workstation may write to the fileblocks. Other workstations can continue to read from the fileblocks and they may also call <BTReadLockAllOpenFileBlocks> themselves. <BTUnlockAllOpenFileBlocks> removes read locks from all the fileblocks just as it would remove level 1 or 2 locks.

There are no deadlock problems possible with read locks. Note that it is possible to call <BTLockAllOpenFileBlocks> after calling <BTReadLockAllOpenFileBlocks> without calling <BTUnlockAllOpenFileBlocks> in between.

Level 6: <BTReadLockFileBlock> and <BTUnlockFileBlock>

This locking level is just like level 5 except that it affects only a specified network fileblock. B-Tree Filer uses temporary level 6 locks frequently to guarantee that the fileblock remains stable while it completes a particular operation.

General Comments About Locking

Almost all methods of writing to a network fileblock require that it first be locked at level 1 or 2. The following routines will fail unless a degree 1 or 2 lock is in force:

```
<BTPutRec>  
<BTAddRec>  
<BTDeleteRec>  
<BTAddKey>  
<BTDeleteKey>
```

(<BTPutRec> may in some circumstances be used to update a fileblock without a level 1 or 2 lock. See Chapter 5 for details.)

The mechanism used to lock a fileblock can be described as follows:

- for a level 1 or 2 write lock, the first byte of the fileblock's dialog file is physically locked. This lock fails if another workstation already has the fileblock locked for write access.

- for a level 1 or 2 write lock, a second section of the dialog file is also physically locked. The size of this section is proportional to the maximum numbers of workstations and indexes for which B-Tree Filer is configured. This lock fails if another workstation already has the fileblock locked with a level 5 or 6 read lock.
- for a level 5 or 6 read lock, one workstation's worth of the dialog file's second section is locked. This lock will fail if another workstation already has the fileblock locked for writing, but it will not fail if another workstation also has the fileblock locked for reading.
- for a level 3 record lock, the first four bytes of the physical record in the data file are locked.

Procedures that access a network fileblock (with three exceptions, see below) always read the dialog file and thereby recognize the locks.

Since the data file is not directly affected by locks of degree 1, 2, 5, or 6, it is possible for another workstation to circumvent the lock and read or even modify a record. <BTGetRec> with a parameter <ISOLock> of True will read a fileblock locked at level 1 or 2. <BTPutRec> with a parameter <ISOLock> of True will modify a record even if the fileblock is locked at level 1 or 2. Of course, these routines must be used with great care.

<BTGetRecReadOnly> will read from a locked fileblock even if the record itself has been locked at level 3. It cannot read the first four bytes of the record if those are locked. However, if the data record reserves the first four bytes for a deletion marker, no real information is lost.

It is important to note that if record locks of level 3 or 4 are used by one workstation, another station will fail to access the locked record even if it has previously locked the entire fileblock using level 1 or 2. In most cases we recommend avoiding record locks altogether since they complicate application design.

E. Converting Single User Programs

It is easy to convert an existing single user program for network capabilities by using level 1 locking exclusively.

- Call <BTLockAllOpenFileBlocks> before every write access, and call <BTUnlockAllOpenFileBlocks> afterwards. Check that the locks were successfully placed and retry if they were not.
- Add checks after every B-Tree Filer call to handle the additional errors that may occur (lock violations, etc.) and retry when appropriate.
- Follow two rules for performance optimization: do not perform any keyboard input while a lock is active; and perform as many B-Tree Filer calls as possible within each lock.

While this approach is straightforward, it does not provide optimum performance when a program has several unrelated fileblocks open simultaneously. For this reason individual fileblock locks (level 2) are often a better choice, but not without added complication. Additional steps must be taken to prevent deadlock if level 2 locks are used. Consider the following scenario:

- Workstation 1 successfully places a lock on fileblock A.
- Station 2 successfully places a lock on fileblock B.
- Station 1 now attempts to lock fileblock B. Since that is not possible, station 1 continually requests the lock.
- Station 2 now attempts to lock fileblock A. Since that is not possible, station 2 continually requests the lock.
- Both stations will wait indefinitely for the other to give up its lock on the other fileblock.

The best solution is to limit the number of retries. When a station reaches the limit, it should give up and release all related locks it has already been granted. It should then wait for a quasi-random amount of time before trying again. The delay may be provided by requiring an operator to request a transaction again, or by using the Random function to generate a non-interactive program delay. In this way, one of the workstations will gain a lead on the other and beat the deadlock.

Programs using locks of level 3 or 4 must account for additional locking conflicts:

- The same deadlock just described for entire fileblocks may occur with single records. This can be solved the same way.

- A level 1 or 2 lock of an entire fileblock no longer guarantees that a workstation can access any particular existing record, since another station may have a lock of degree 3 or 4 on that record. This can be solved by setting the <ISOLock> parameter of <BTGetRec> and <BTPutRec> to True.

Further difficulties not directly related to locking may arise in the network environment. Consider the following three scenarios:

1. A record that was edited by workstation 1 is deleted or changed by station 2 in the meantime.
2. A record that is to be deleted by workstation 1 is deleted or changed by station 2 in the meantime.
3. A listing being printed by workstation 1 is made inaccurate when station 2 modifies the index during the listing. (For example, if station 2 modifies the index of a record already printed, and reindexes the record to a position beyond the current end of the listing, that record will be printed twice.)

The simplest solution to these problems is obtained simply by locking the entire fileblock throughout the process of editing, deletion, or listing. This solution is rarely adequate, however, since other workstations are so frequently unable to access the data.

A better idea is to lock just the single record being edited or deleted. Even this will interrupt a listing being performed by another station, unless the listing routine uses <BTGetRecReadOnly> to read the records.

By using the following algorithms, we can avoid locking even the individual record while interactive entry occurs.

Editing

- Read the record
- Interactively perform the edit
- Lock the record
- Reread the record
- Compare with the original record
- If identical
 - Lock the fileblock
 - Write back the modified record
 - If a key was modified
 - Delete the old keys
 - Add the new keys
 - Unlock the fileblock
 - Unlock the record
- If not identical (another station modified it)
 - Unlock the record
 - Present the new data to the user and perform the edit again

Deleting

- Read the record
- Interactively obtain confirmation to delete
- Lock the record
- Reread the record
- Compare with the original record
- If identical
 - Lock the fileblock
 - Delete the record and all keys
 - Unlock the fileblock
 - Unlock the record
- If not identical (another station modified it)
 - Unlock the record
 - Start from the beginning

If record locks are not used elsewhere in an application, it's easy to avoid them here too. The simplest way is to move the fileblock lock up a few steps to the position of the record lock. A read lock may also be used to reduce the time while other stations can't read from the fileblock. Consider the following slightly modified versions of the algorithms (modified lines are indicated by asterisks).

Editing

```
Read the record
Interactively perform the edit
Read lock the fileblock **
Reread the record
Compare with the original record
If identical
    Lock the fileblock
    Write back the modified record
    If a key was modified
        Delete the old keys
        Add the new keys
    Unlock the fileblock
**
If not identical (another station modified it)
    Unlock the fileblock **
    Present the new data to the user and perform the edit again
```

Deleting

```
Read the record
Interactively obtain confirmation to delete
Read lock the fileblock **
Reread the record
Compare with the original record
If identical
    Lock the fileblock
    Delete the record and all keys
    Unlock the fileblock
**
If not identical (another station modified it)
    Unlock the fileblock **
    Start from the beginning
```

Read locks are also valuable in solving difficulty 3 described above. By placing a read lock during the generation of a critical report, an application allows other users to continue reading from the fileblock, but prevents anyone from corrupting the reporting by modifying the fileblock.

Examples in the following section (as well as in the NETDEMO program) demonstrate working implementations of these algorithms.

F. Network Example

The examples already provided in Section 4.C are largely applicable here as well. Please refer to that section if you haven't already read it.

In this section, we will offer code examples for dealing with some of the thorny issues discussed in the previous section. The source code can be found in NETEXAMP.PAS.

Reacting to Locks

Consider the function FindRecord from Section 4.C. There are several ways to make this routine network-aware. One can simply add appropriate error handling after calling the existing routine:

```
if not FindRecord(P, RefNr, KeyNr, Key) then
  if IsamErrorClass = 2 then begin
    {Lock error}
    Writeln('File or record locked');
    {Abort operation}
  end;
```

This is simple, but not good for much. There is not even an attempt to retry the operation. In some cases, the operating system can be programmed to retry automatically. (See <SetDosRetry> in Chapter 5.) Unfortunately, this procedure is not functional for all networks. Even when it is, the application must still handle the case when a lock error occurs after the retries run out.

Therefore it's clear that we need a new routine that will retry in case of a lock, and eventually either give up or prompt the user for further direction. The following code demonstrates this concept:

```
const
  MaxRetries = 10;
  RetryCount : Integer = 0;

function IsLockError(Ask : Boolean) : Boolean;
begin
  if IsamOK or (IsamErrorClass <> 2) then begin
    {No error, or non-locking error}
    IsLockError := False;
    {Reset retry count}
    RetryCount := 0;
  end else begin
    {Lock error}
    IsLockError := True;
    inc(RetryCount);
    if RetryCount > MaxRetries then begin
      {Out of retries}
      if not Ask then
```



```

        {Abort the operation without even asking}
        IsLockError := False
    else if not YesNo('Lock error. Try again?') then
        {Abort the operation if the user says to do so}
        IsLockError := False;
        {Reset retry count}
        RetryCount := 0;
    end;
end;
end;
end;

```

YesNo is a routine that prompts the user for a yes/no response.

IsLockError may be called instead of polling <IsamOK> after each B-Tree Net operation. If IsLockError returns False, and <IsamOK> is also False, then a non-locking error has occurred (or perhaps the retry limit was exceeded and the user chose to abort the operation).

With IsLockError in place, we can rewrite FindRecord as follows:

```

function FindRecord(var P : PersonDef;
                    var RefNr : LongInt;
                    KeyNr : Integer;
                    var Key : IsamKeyStr) : Boolean;
begin
    FindRecord := False;
    repeat
        BTSearchKey(PF, KeyNr, RefNr, Key);
    until not IsLockError(True);
    if not IsamOK then begin
        {Key not found, program error, or abortive locking error}
        Exit;
    end;
    repeat
        BTGetRec(PF, RefNr, P, False);
    until not IsLockError(True);
    if not IsamOK then begin
        {Error handling}
        Exit;
    end;
    FindRecord := True;
end;

```

This example can be extended to all other reading functions from Section 4.C.

Reacting to Modified Data

In a single user application, one can be sure that once a record or key is entered, it is always available until the application itself modifies or deletes it. This is not the case in a network application. For example:

```

workstation 1 reads record number 10 and displays a "delete?" prompt
workstation 2 deletes record number 10
workstation 1's user accepts the prompt
workstation 1 deletes record number 10

```

Record 10 is deleted twice--an action that should not be performed on the data file. A single user solution was already presented in Section 4.C. A corresponding network implementation of the function DeleteRecord can be written as follows. As in the previous examples, we rely on the first field of each data record for a deletion tag.

```

var
  MaxError : Integer;

procedure SetMaxError;
var
  ErrorClass : Integer;
begin
  ErrorClass := IsamErrorClass;
  if ErrorClass > MaxError then
    MaxError := ErrorClass;
end;

procedure UpdateUnlock(RefNr : LongInt);
begin
  if BTFileBlockIsLocked(PF) then begin
    BTUnlockFileBlock(PF);
    if not IsamOK then
      {Hardware failure? shouldn't happen}
      SetMaxError;
    end;
  if BTRecIsLocked(PF, RefNr) then begin
    BTUnlockRec(PF, RefNr);
    if not IsamOK then
      {Hardware failure? shouldn't happen}
      SetMaxError;
    end;
  end;
end;

function UpdateLock(RefNr : LongInt) : Boolean;
begin
  UpdateLock := False;
  {Record locking is needed only if the application uses
   record locks elsewhere}
  BTLockRec(PF, RefNr);
  if not IsamOK then begin
    {Record could not be locked}
    SetMaxError;
    Exit;
  end;
  {Lock the fileblock}
  LockFileBlock(PF);
  if not IsamOK then begin
    {FileBlock could not be locked}
    SetMaxError;
  end;
end;

```

```

        UpdateUnlock(RefNr);
        Exit;
    end;
    UpdateLock := True;
end;

function MatchRecord(P1, P2 : PersonDef) : Boolean;
begin
    {Return true if P1 and P2 are the same}
    MatchRecord := True;
end;

function DeleteRecord(P : PersonDef; RefNr : LongInt) : Integer;
var
    KeyNr : Integer;
    TempP : PersonDef;
begin
    {At this point, the record to be deleted has already been read into
    record P, and the user has verified that a deletion is to occur.}

    {MaxError is the highest error class encountered}
    MaxError := 0;

    {Lock the record and the fileblock}
    if not UpdateLock(RefNr) then begin
        {Couldn't lock}
        DeleteRecord := MaxError;
        Exit;
    end;

    {Get the record again to see if it still matches the original}
    BTGetRec(PF, RefNr, TempP, False);
    if not IsamOK then begin
        {Shouldn't happen}
        SetMaxError;
        UpdateUnlock(RefNr);
        DeleteRecord := MaxError;
        Exit;
    end;

    if TempP.Del <> 0 then begin
        {Record was already deleted. That's ok since we wanted to delete it also}
        UpdateUnlock(RefNr);
        DeleteRecord := MaxError;
        Exit;
    end;

    if not MatchRecord(P, TempP) then begin
        {Record was modified in the meantime. Return a "warning" error class}
        MaxError := 1;
        UpdateUnlock(RefNr);
        DeleteRecord := MaxError;
        Exit;
    end;

    {Finally perform the deletion}
    for KeyNr := 1 to BTNrOfKeys(PF) do begin

```

```

BTDeleteKey(PF, KeyNr, RefNr, CreateKey(P, KeyNr));
if not IsamOK then
  if IsamError = 10220 then
    {Key already deleted. Shouldn't happen, but it's still ok if so}
  else begin
    {Error handling}
    SetMaxError;
    UpdateUnlock(RefNr);
    DeleteRecord := MaxError;
    Exit;
  end;
end;
BTDeleteRec(PF, RefNr);
if not IsamOK then
  SetMaxError;

{Unlock the record and the fileblock}
UpdateUnlock(RefNr);
DeleteRecord := MaxError;
end;

```

The UpdateLock routine locks the record to be updated, then locks the entire fileblock so that keys can be removed safely. Note that the record lock is necessary only if other portions of the same application are using record locks for independent reasons. UpdateUnlock unlocks the record and fileblock. MatchRecord compares two data records and returns True if they are considered to be the same. An actual implementation of this routine will typically compare each field of the two records and return True only if all of them are the same.

The global variable MaxError keeps track of the worst error class encountered and DeleteRecord returns that error class. A return value of 0 means that the routine succeeded. Error class 1, normally a warning for a key not found, is used in DeleteRecord to indicate that another workstation has modified the record to be deleted. If DeleteRecord returns 1, the application should warn the user, redisplay the record, and determine whether the record should still be deleted. Error class 2 is a locking error, as usual, and means that another station has the fileblock or the individual record locked. An appropriate response is to delay for a short time and retry for a specified number of times. Error classes 3 and 4 mean severe errors.

The function ModifyRecord can be implemented similarly. All of the example routines, suitably modified for network use, are provided in NETEXAMP.PAS.

G. Questions and Answers

In this section we'll discuss some of the most common application design questions that arise for B-Tree Filer.

Save Mode or Normal Mode?

When a fileblock is opened with <BTOpenFileBlock> a boolean parameter specifies whether "save mode" is to be used. In save mode, data integrity is increased by writing portions of the fileblock to be modified to the dialog file before the changes are made. Then, if an error occurs, the fileblock can be restored to its known state prior to the operation. If no error occurs, all new data is flushed to the disk before the backup information is removed from the dialog file. If the system crashes before an operation is complete, B-Tree Filer automatically repairs the fileblock by using the information in the dialog file the next time the fileblock is opened.

As you can imagine, save mode reduces performance when adding, deleting, or modifying information in the data and index files. Therefore you should consider the following factors before choosing to use save mode.

- You can call <BTFlushFileBlock> at specific times to guarantee that all information is flushed from memory onto the disk storage device.
- The routines in the REBUILD and VREBUILD units will reconstruct a corrupted index file from scratch. Reconstruction of 5000 records with four keys per record takes 5 to 20 minutes.
- B-Tree Filer maintains an internal flag that indicates when a fileblock wasn't correctly closed after being modified (see <BTForceWritingMark> in Chapter 5). When an application attempts to open such a fileblock, an error is generated and the application can trigger an automatic rebuild.
- The performance loss due to save mode is usually reduced on a network because of smart caching on the file server. (However, see procedure <BTForceNetBufferWriteThrough> in Chapter 5.)
- On Novell networks, B-Tree Filer can work with the transaction tracking services (TTS) of the operating system. This provides another approach to data integrity. See Section 7.A.

The decision between save mode and normal mode for a single user application is usually made by comparing the day to day performance loss for save mode against the time to reconstruct the database if a severe error does occur. For a network application, the data sets are usually larger and more critical, leading to higher costs if the data is lost for any period of time. As a result, we often recommend save mode for network applications when Novell's TTS is not available. If save mode seems too

slow, use normal mode and provide for a regular backup (perhaps every few hours) of the files. Some networks even support automatic backup to streaming tape so that data saving can take place in the background without disrupting the system.

The final choice between save mode and normal mode will be based on a combination of measurable and subjective factors. Are updates done in batch or interactive mode? How does it "feel" to add and delete records interactively under heavy network load? How disruptive is a rebuild if a crash occurs? What backup strategy is already in place for the network? Fortunately, just by toggling one boolean parameter to <BTOpenFileBlock>, you can easily experiment with your application.

Network Emulation

There are three ways to control how a fileblock is opened for network use:

1. Define NoNet or one of the true network interfaces in BTDEFINE.INC.
2. Pass <NoNet> or one of the true network types to <BTInitIsam>.
3. Pass False or True for the <Net> parameter when opening a fileblock with <BTOpenFileBlock>.

If BTDEFINE.INC is set for NoNet (required for the single user version of B-Tree Filer), then no fileblock can be opened for network use and no network emulation is available. If BTDEFINE.INC defines one or more network interfaces and <NoNet> is passed to <BTInitIsam>, then B-Tree Filer enters "network emulation mode". The behavior of a fileblock opened with <Net> set to True differs depending on whether network emulation is active or not. The following table summarizes the differences:

	NoNet compilation	Network Emulation
Handles per fileblock	2	3
Write accesses	locks not required	fileblock lock required
Probability code will work correctly on a true network	medium	high
Execution speed compared to a true network	much higher	noticeably higher
Effect of read locks on speed of a series of read accesses	no effect	>2x faster

See the documentation for <BTReadLockFileBlock> in Chapter 5 for more information about the last point.

5. B-Tree Filer Identifiers

This chapter describes the identifiers interfaced by the FILER unit of B-Tree Filer. The functions and procedures are arranged alphabetically. Appendix B contains a list of routines arranged by functional category.

Some interfaced identifiers are not described here. Those are intended for the internal use of the unit, and are interfaced for communication with the other units provided with B-Tree Filer. See the source code for details.

Note that all routines described in this chapter work with single user as well as network fileblocks. A few prerequisites must be met before they will work correctly in a multi-user, network environment:

1. A network directive must have been chosen at compile time to select a network interface. Details can be found in Section 2.A.
2. If the network itself cannot generate a unique workstation number, the variable `<IsamWSNr>` must be assigned a valid number before calling `<BTInitIsam>`. See Section 2.F and the procedure `<WSNrLogIn>` from the ISAMTOOL unit for further details.
3. When the procedure `<BTInitIsam>` is called to initialize the FILER unit, the parameter `<ExpectedNet>` must have a value other than `<NoNet>`. See the entry for `<BTInitIsam>` for further details.
4. Finally, network fileblocks must be opened by calling `<BTOpenFileBlock>` with the `<Net>` parameter set to True. New fileblocks are created by calling `<BTCreatFileBlock>`, then opened with `<BTOpenFileBlock>` in the same fashion.

Because the routines described in this chapter aren't really standalone routines--they often interact with each other closely--it isn't always possible to provide a meaningful short example. If no example is provided for a routine, you should look at Sections 4.C and 4.F where the programming examples show many of these functions used together.

Not all examples contain the retry loops required for locking errors on a network. We do this for clarity and to prevent the manual from eating too many more trees. General recommendations for dealing with locking errors can be found in Section 4.F.

Note that all user-configurable constants of the FILER unit are contained in the source file FILER.CFG. Edit this file if you want to change compile-time constants.

A. Constants

AddNullKeys : Boolean = True;

This typed constant affects the behavior of only the REORG, VREORG, REBUILD, and VREBUILD units. When <AddNullKeys> is True, all keys returned by the user-defined BuildKey routine--even empty strings--are added to the fileblock as it is reorganized. If <AddNullKeys> is set to False, these units won't add empty keys to a reorganized fileblock. This provides an extra degree of flexibility for Fileblocks that don't store keys for every index and record.

```
DatExtension : String[3] = 'DAT';  
IxExtension : String[3] = 'IX';  
DiaExtension : String[3] = 'DIA';  
SavExtension : String[3] = 'SAV';  
MsgExtension : String[3] = 'MSG';
```

File names passed to the B-Tree Filer routines will use these standard extensions. Note that these strings will override any extensions that you've specified when opening a fileblock. <DatExtension> is applied to the data file, <IxExtension> to the index file, <DiaExtension> to the dialog file, <SavExtension> to the copy of the data file that <RebuildFileBlock> and <ReorgFileBlock> create, and <MsgExtension> to the message file that the rebuild routines create when duplicate keys are detected.

```
IsamDelayBetwLocks : Word = 50; {milliseconds}  
IsamRetriesForLock : Word = 40; {tries}
```

In a network environment, B-Tree Filer will automatically retry certain locking calls. <IsamDelayBetwLocks> is the approximate number of milliseconds to delay between retries and <IsamRetriesForLock> is the number of retries to perform before the lock attempt fails and returns with an error code. In a very active network, <IsamRetriesForLock>'s standard value of 50 might be too small and could be increased to at most 100. Note that delays generated within B-Tree Filer have an internal resolution of about 55 milliseconds.

Automatic retries apply only during calls to <BTLockFileBlock> and <BTLockAllFileBlocks>, which calls <BTLockFileBlock>. <BTLockFileBlock> applies two physical locks in succession (see Section 4.D). If the first lock fails, it means that another workstation already has the fileblock locked for writing; in this case, <BTLockFileBlock> exits immediately with error 10330. If the first lock succeeds and the second lock fails, it may mean that B-Tree Filer running on another workstation has used a temporary internal read lock. These read locks are used frequently to assure stability of the fileblock while a sequence of required index pages are read. In this second case, Filer will automatically retry the lock. If the lock cannot be obtained after the retries, <BTLockFileBlock> will exit with error 10330. The application may of course call <BTLockFileBlock> repeatedly in its own retry loop.

IsamFileNameLen = 64;

Maximum length of any single drive, directory, and filename for a fileblock. Note that this value is less than the DOS limit of 79 characters. You may change <IsamFileNameLen> without rebuilding fileblocks. Three arrays of size <IsamFileNameLen>+1 are contained within each variable of type <IsamFileBlock>.

IsamFlushDOS33 : Boolean = True;

Affects the behavior of the low level <IsamFlush> procedure, which is called whenever B-Tree Filer needs to assure that the contents of DOS file buffers are flushed to disk. Flushing occurs whenever the application calls <BTFlushFileBlock> or <BTFlushAllFileBlocks>. It also occurs when the application has enabled "write-through" by calling <BTForceNetBufferWriteThrough> with a parameter of True (see that procedure for details).

The <IsamFlush> procedure uses any of three mechanisms to flush a file:

1. If <IsamFlushDOS33> is True and the DOS version is at least 3.3, <IsamFlush> calls DOS function \$68, which is preferred over alternative methods because it will never fail for lack of a free file handle and it doesn't disturb any locks that may be in place. Unfortunately, when running under PC-MOS, function \$68 is accepted with no error, but no actual flush occurs. This is the only known case of function \$68 problems, but other lesser networks may have the same behavior; hence the typed constant <IsamFlushDOS33> which you can use to disable this method of flushing. If flush method 1 is not used, or if it fails, flush method 2 is tried.
2. <IsamFlush> duplicates the file handle to be flushed (by calling DOS function \$45) and then closes the duplicate handle. This sequence will fail if no free file handle is available. Unfortunately, certain networks, including all SHARE-based LANs, remove locks when closing the duplicated handle. Other networks, including Novell and CBIS, leave the locks in place. If flush method 2 fails for lack of a file handle, and if the file is not part of a network fileblock, <IsamFlush> moves on to method 3.
3. <IsamFlush> closes the handle and reopens it. This technique is not even attempted for a network file because all networks, with the exception of Novell, remove locks from the closed file. If none of the methods work, <IsamFlush> returns an error code of 10150.

You will conclude that, under many circumstances, there are severe problems with flushing locked fileblocks. If your target environment falls into this category, be sure not to call <FlushFileBlock> or <FlushAllFileBlocks> while locks are in place on the data file, and be sure not to enable write-through at all.

MaxHeight = 8;

Specifies the deepest level of the B-tree. When <BTreeInitIsam> is called, free heap space must exist for at least <MaxHeight> index pages. (The heap space can be allocated from EMS memory if EMS is available and B-Tree Filer is configured correctly. See Section 2.B and the <BTreeInitIsam> procedure in this chapter.) <MaxHeight> and <PageSize> together determine the maximum number of keys that may be added to any index. In general we recommend that you do not modify <MaxHeight>. If you need to do so, keep the following rules in mind:

1. The smallest acceptable value for <MaxHeight> in B-Tree Filer is 4, regardless of how few keys will be added to the fileblock. Worst case page balancing and splitting during a call to <BTreeAddKey> requires 4 index pages to be in memory at once.
2. The B-tree manager will not detect an error if you exceed the maximum number of keys allowed by <MaxHeight> and <PageSize>. Due to the dynamic nature of page balancing, error detection would require unacceptable performance overhead. It is your responsibility to choose values that work with your application.
3. The algorithm for determining the capacity of a B-tree can be summarized as follows, where Cap(i) is the capacity of level i of the tree:

```
Cap(1) = 1
Cap(2) = PageSize
Cap(i) = Cap(i-1)*(1+PageSize div 2)    (i >= 3)
```

and the total capacity is the sum of the capacities for each level.

For the default value of <PageSize>=62, the following total capacities are computed:

MaxHeight	Maximum Keys
4	65,535
5	2,097,151
6	67,108,863
7	2,147,483,647 (= 2 ³¹ = MaxLongInt)
8	68,719,476,735 (> 2 ³²)

As a result, increasing the default value of 8 is meaningless, while reducing the value has to be done with the utmost caution and understanding. A rebuild of the index files after changing this constant is not required.

MaxKeyLen = 30;

<MaxKeyLen> determines the maximum allowed length of a key in any fileblock. Legal values are between 1 and 255. Note that <MaxKeyLen> does not affect the size of the index file (whose size is determined by key lengths specified in an <IsamIndDescr> variable when the fileblock is created), but that <MaxKeyLen> does affect the size of an index page in memory and therefore the number of index pages that will fit into a given amount of heap space.

`MaxNrOfKeys = 100;`

<MaxNrOfKeys> defines the maximum number of indexes in any one fileblock. Legal values lie between 1 and 254. Increasing this constant increases proportionally the amount of storage needed for a variable of type <IsamIndDescr> but has no effect on index file size.

`MaxNrOfWorkStations = 50;`

<MaxNrOfWorkStations> contains the maximum number of workstations used in the network. The default value of 50 will suffice for typical network installations. Legal values range from 1 to 65534, but unnecessarily increasing the value will increase the access time for all network operations.

`MinimizeUseOfNormalHeap = $40000000;`

When this value is added to the <Free> parameter passed to <BTInitIsam>, B-Tree Filer will use the least possible normal heap space for page buffers. See <BTInitIsam> for further information. This constant should not be changed.

`PageSize = 62;`

Specifies the maximum number of keys in a B-tree page. The minimum amount of heap space required when calling <BTInitIsam> is directly related to <PageSize> and <MaxHeight>. The results of many test runs indicate that the default value of 62 is optimal; therefore we recommend that you do not change it. If you would like to modify the default, note the following restrictions:

1. <PageSize> must be an even number between 14 and 246.
2. After changing <PageSize>, all index files must be rebuilt.
3. The best values are even numbers just below a power of two, i.e., 14, 30, 62, 126. The only desirable alternative to 62 is 30, whose data retrieval performance in a network environment is comparable to that of 62.

`SearchForSequentialDefault : Boolean = True;`

Default state of the special searching method applied to <BTNextKey> and <BTPrevKey> operations. This method is useful in recovering from "sequential access not allowed" errors that arise when multiple workstations can modify a fileblock. The default value can also be changed for a single index with the procedure <BTSetSearchForSequential>.

`VersionStr = '05.21';`

Defines the current version number of Filer where it can be accessed by an application program.

B. Types

```
IsamFile = record
  Handle : Word;
  Name   : array[0..IsamFileNameLen] of Char;
end;
```

Type describing a single open file. This is used internally by B-Tree Filer and passed to lower level routines such as <IsamAssign>, <IsamRewrite>, and <IsamFlush>. A variable of type <IsamFileBlock> contains three <IsamFile> fields: one each for data file, index file, and dialog file. <Handle> is a valid handle number assigned by DOS, or \$FFFF when the file is closed. <Name> is the null-terminated pathname of the file.

```
IsamFileBlockName = String[192];
```

All filenames passed to <BTCreatFileBlock>, <BTOpenFileBlock>, <Rebuild(V)FileBlock>, <Reorg(V)FileBlock> and <FixToVarFileBlock> must of this type. Variables of this type must contain one, two, or three valid DOS filenames when passed to one of the above routines. Drive name and path are optional. The current directory is used as a starting point if a path doesn't begin with a '\'. No extensions should be given. Extension are automatically appended, using the constants defined above.

If multiple filenames are given, they should be separated by a semicolon (;).

The first filename determines the drive, directory, and name of the data file and also of the dialog file for a network or save-mode fileblock. The second filename specifies the name and location of the index file. The third filename is only useful for the procedures <Rebuild(V)FileBlock>, <Reorg(V)FileBlock>, and <FixToVarFileBlock>, where it specifies the name and location of the saved copy of the data file. The third name may be passed to the other routines (e.g. <BTOpenFileBlock>), but it will be ignored there. Examples of valid names are:

```
'ADDRESS'
Data file  'ADDRESS.DAT'
Index file  'ADDRESS.IX'

'C:\FILES\ADDRESS'
Data file  'C:\FILES\ADDRESS.DAT'
Index file  'C:\FILES\ADDRESS.IX'

'C:\FILES\ADDRESS;D:\INDEX\ADDRESS'
Data file  'C:\FILES\ADDRESS.DAT'
Index file  'D:\INDEX\ADDRESS.IX'
```

We don't recommend the following naming convention, even though B-Tree Filer will accept it:

```
'C:\FILES\ADDRESS;D:\INDEX\ADINDEX'
Data file  'C:\FILES\ADDRESS.DAT'
Index file  'D:\INDEX\ADINDEX.IX'
```

In this case the logical connection between the data file and the index file is no longer apparent.

```
IsamFileBlockPtr = ^IsamFileBlock;
```

A variable of this type points to information describing each opened fileblock. You can think of it as a pointer to a file variable. An `<IsamFileBlockPtr>` is allocated and initialized with a call to `<BTOpenFileBlock>`. All further operations on the fileblock are performed by passing this variable to a B-Tree Filer routine. You needn't be concerned with the contents of an `<IsamFileBlock>` record, except to note that an opened fileblock consumes about 300-500 bytes of heap space depending on the number of indexes and whether it incorporates network support. See `<BTOpenFileBlock>` for more information.

```
IsamFileName = String[IsamFileNameLen];
```

A string defining the drive, directory, and name of a single file. Parameters of this type are passed to lower level B-Tree Filer routines such as `<IsamExists>` and `<IsamAssign>`.

```
IsamIndDescr = array[1..MaxNrOfKeys] of
  record
    KeyL : 1..MaxKeyLen;
    AllowDupK : Boolean;
  end;
```

A variable of this type is passed to `<BTCreatFileBlock>` to describe the indexes associated with the fileblock. `<IsamIndDescr>` describes the length and type for each index. Key length may be from 1 to `<MaxKeyLen>` and is stored in the `<KeyL>` field. Keys can be of two different types, depending on whether or not duplicate keys may occur:

1. **Primary Keys:** `AllowDupK = False`. An example is a customer number that is guaranteed to be unique. Zero, one, or more primary keys may be defined for a fileblock. Remember that `<BTAddKey>` will not accept duplicate primary keys.
2. **Secondary Keys:** `AllowDupK = True`. Examples include last name and zip code, which are not guaranteed to be unique. `<BTAddKey>` will successfully add duplicate secondary keys as long as the data reference number for two such keys does differ.

```
IsamKeyStr = String[MaxKeyLen];
```

All key strings used to perform index operations must of this type.

```
NetSupportType = (NoNet, Novell, MsNet, MsNetMachName, CBISNet,  
PCMO386, VinesNet, NTNXNet, DesqView);
```

This type enumerates the various networks that B-Tree Filer supports. The procedure <BTInitIsam> requires a parameter of this type to specify the current network environment. Upon successful initialization of the network, the function <BTNetSupported> returns the same value.

C. Variables

IsamOK : Boolean;

This variable is set by every B-Tree Filer routine. Upon entry to each routine, <IsamOK> is initialized to True, even if a previous error was pending. If <IsamOK> remains True when the routine returns, the requested operation was successfully completed. Otherwise, the variable <IsamError> contains a code corresponding to the error that occurred. <IsamError> should never equal zero when <IsamOK> is False. <IsamOK> should be checked after each call to a B-Tree Filer routine.

IsamError : Integer;

This variable is set by every B-Tree Filer routine. If <IsamOK> is False, <IsamError> corresponds to the type of error that occurred. A complete list of error codes is given in Appendix A.

IsamWSNr : Word;

This variable must contain a unique number between 1 and <MaxNrOfWorkStations> for each workstation in a network environment. For many networks, such as Novell, <BTInitIsam> will automatically initialize <IsamWSNr> by requesting information from the network. For other networks that do not provide an appropriate service, it is the application's responsibility to initialize <IsamWSNr> before calling <BTInitIsam>. See Section 2.F for more information.

IsamReXUserProcPtr : Pointer;

The FILER unit sets this variable to Nil within its initialization block. When an application sets <IsamReXUserProcPtr> to a non-Nil value, the pointer must contain the address of a procedure that will be called by <Rebuild(V)FileBlock>, <Reorg(V)FileBlock>, and <FixToVarFileBlock> in the (V)REBUILD, (V)REORG, and FIXTOVAR units respectively. The procedure will be called for each record and key that is processed. It may be used to display status information, compute statistics, or allow the user to abort the rebuild. The routine must be declared far, must be global, and must match the following prototype declaration:

```

($F+) (Also must be global)
procedure UserStatusRoutine(KeyNr : Integer;
    NumRecsRead,
    NumRecsWritten : LongInt;
    var Data;
    Len : Word);

begin
    (display status...)
end;
($F-)

```

<KeyNr> is the index number currently being added. <NumRecsRead> is the number of records read up to this point and <NumRecsWritten> is the number of records written. <Data> is the data record being worked on, and <Len> is the number of bytes of data in the record. The rebuild routines all work by first copying each valid data record to a new data file. During this pass, UserStatusRoutine will be called once for each record with <KeyNr> set to zero. After the data has been copied, all keys for index number one (<KeyNr>=1) are added and UserStatusRoutine is called for each key, then all keys for index number two, and so on. See Chapter 6 for more information.

```
IsamCompiledNets : Set of NetSupportType;
```

The initialization block of the FILER unit assigns this variable its value. The set contains all network types activated within BTDEFINE.INC which are therefore valid options to pass to <BTInitIsam>. This variable should be considered read-only and should not be changed.

```
IsamDOSFunc : Word;
```

This variable contains the function code for the last DOS function called by B-Tree Filer. (The function code is the value passed in the AX register for the DOS int \$21 call.) If an error occurred while making the DOS function call, the value of <IsamDOSFunc> retains its value all the way to the end of the B-Tree Filer routine. Hence, <IsamDOSFunc> may be used to determine exactly which DOS function produced a particular <IsamError>. <IsamDOSFunc> has a value of 0 if the routine made no calls to a DOS function.

```
IsamDOSError : Word;
```

This variable contains a non-zero value if there was a DOS error during a call to a B-Tree Filer routine. The value is the one returned by DOS in the AX register.

D. Procedures and Functions

Almost all of the following routines require the parameter <IFBPtr> of type <IsamFileBlockPtr>. Its use is to differentiate between the different fileblocks opened by <BTOpenFileBlock>, just as you would pass a different file variable to a Turbo Pascal Read or Write routine. <BTOpenFileBlock> is the only routine that allocates and initializes a variable of type <IsamFileBlockPtr>. <BTCloseFileBlock> deallocates the fileblock variable and sets the <IFBPtr> to Nil.

The parameter <IFBPtr> will not be explained again in the descriptions that follow.

Some interfaced procedures and functions of the FILER unit are not described in detail here. These routines are interfaced primarily for the use of other B-Tree Filer units such as REORG and VREORG. If you are writing similar utility units for B-Tree Filer files, you may find the following routines useful. Note that these routines return status information in <IsamOK> and <IsamError> just like all other FILER routines. See the source code for more information.

```
function BTIsamLockRecord(Start, Len : LongInt; Handle : Word) : Boolean;
  (-Locks the bytes <Start> to <Start>+<Len>-1 of the file
   with handle <Handle> using the active network locking call)

function BTIsamUnlockRecord(Start, Len : LongInt; Handle : Word) : Boolean;
  (-Unlocks the bytes <Start> to <Start>+<Len>-1 of the file
   with handle <Handle> using the active network unlocking call)

procedure IsamAssign(var F : IsamFile; FName : IsamFileName);
  (-Assigns <FName> to the file <F>)

procedure IsamBlockRead(var F : IsamFile; var Dest; Len : Word);
  (-Reads a block of data with <Len> bytes out of the file <F> to <Dest>)

procedure IsamBlockReadRetLen(var F : IsamFile; var Dest; Len : Word;
                               var BytesRead : Word);
  (-Reads a block of data and returns the number of bytes read)

procedure IsamBlockWrite(var F : IsamFile; var Source; Len : Word);
  (-Writes a block of data with <Len> bytes to the file <F> from <Source>)

procedure IsamClearOK;
  (-Resets all status variables, even internal ones)

procedure IsamClose(var F : IsamFile);
  (-Closes the file <F>)

procedure IsamCopyFile(Source, Dest : IsamFileBlockName;
                       DeleteSourceAfterCopy : Boolean);
  (-Copies file <Source> to <Dest>)

procedure IsamDelay(MilliSecs : LongInt);
  (-Delays for <MilliSecs> ms using a DOS call (resolution 55ms))

procedure IsamDelete(var F : IsamFile);
  (-Deletes the file <F>)
```

```

function IsamExists(Name : IsamFileName) : Boolean;
  (-Returns True if the specified file exists)

procedure IsamExtractFileNames(FNameComp : IsamFileBlockName;
                               var FNameD, FNameI : IsamFileBlockName);
  (-Separates two file names delimited by ";")

procedure IsamFlush(var F : IsamFile; var WithDUP : Boolean;
                   NetUsed : Boolean);
  (-Flushes the buffers of <F> to disk, depending on the constant
   <IsamFlushDOS33> and the parameter <NetUsed>)

function IsamForceExtension(Name, Ext : IsamFileName) : IsamFileName;
  (-Forces the extension <Ext> onto the file name <Name>)

procedure IsamGetBlock(var F : IsamFile; Ref, Len : LongInt; var Dest);
  (-Reads a block of data with <Len> bytes starting at position <Ref>
   from the file <F> to <Dest>)

procedure IsamLongSeek(var F : IsamFile; Ref : LongInt);
  (-Seeks to the position <Ref> in file <F>)

procedure IsamLongSeekEOF(var F : IsamFile; var Len : LongInt);
  (-Seeks to the end of file in file <F> and returns its length)

procedure IsamPutBlock(var F : IsamFile; Ref, Len : LongInt; var Source);
  (-Writes a block of data from <Source> with <Len> bytes to the file <F>
   starting at position <Ref>)

procedure IsamRename(var F : IsamFile; FName : IsamFileName);
  (-Renames the unopened file <F> to <FName>)

procedure IsamReset(var F : IsamFile; NetUsed, ReadOnly : Boolean);
  (-Opens the file <F> in the specified mode)

procedure IsamRewrite(var F : IsamFile);
  (-Creates the file <F>)

```

All normal procedures and functions in the FILER unit, with the exception of <BTIsamErrorClass>, reset the FILER unit's status variables (<IsamOK> and <IsamError>) as soon as they are called. In some circumstances it may be desirable to retain the previous operation's error codes while making another FILER procedure call. The following routines are available to meet this need. They work just like the corresponding routine without "Peek" in the name, except that they preserve the values of <IsamOK>, <IsamError>, <IsamDOSError>, and <IsamDOSFunc>. However, if another error occurs while in the "Peek" routine itself, that error will be returned instead of any previous error. The "Peek" routines are:

BTPeekNoNetCompiled
BTPeekNetSupported
BTPeekFileBlockIsOpen
BTPeekIsNetFileBlock
BTPeekFileBlockIsLocked
BTPeekFileBlockIsReadLocked
BTPeekGetRecordInfo
BTPeekRecIsLocked
BTPeekRecIsLocked
BTPeekNrOfKeys
BTPeekDatRecordSize
BTPeekKeyRecordSize
BTPeekMinimumDatKeys
BTPeekDataFileName
BTPeekIndexFileName

BTAddKey

Declaration

```
procedure BTAddKey(IFBPtr : IsamFileBlockPtr;  
    Key : Word;  
    UserDatRef : LongInt;  
    UserKey : IsamKeyStr);
```

Purpose

Add a key.

Description

After a new data record has been added with <BTAddRec>, the index file should also be updated. This is done by passing the key string <UserKey> and the data record number <UserDatRef> obtained from the <BTAddRec> call to <BTAddKey>. <BTAddKey> must be called one or more times for every index <Key>, which ranges from 1 to the number of indexes in the fileblock.

<BTAddKey> modifies the internal sequential pointer whether or not the <BTAddKey> call succeeds. Hence, the sequential pointer must be refreshed before calling <BTNextKey> or <BTPrevKey>. The pointer is refreshed automatically if the typed constant <SearchForSequentialDefault> was True when the fileblock was opened or if <BTSetSearchForSequential> has been used to activate the option thereafter. Otherwise, the pointer may be reinitialized by a call to <BTClearKey>, <BTSearchKey>, <BTFindKey>, <BTFindKeyAndRef>, <BTSearchKeyAndRef>, <BTNextDiffKey>, or <BTPrevDiffKey>.

<BTAddKey> can be used with a network fileblock only if the fileblock has been locked for exclusive write access (level 1 or 2).

If <Key> denotes a primary index, <UserKey> must specify a string that does not currently exist in the index. If <Key> is a secondary index, the combination of <UserKey> and <UserDatRef> must still be unique. <BTAddKey> will return with error 10230 if the key is not unique.

Note that all keys must take the form of a string. The supplied NUMKEYS unit provides routines to convert all Turbo Pascal numeric types to and from strings, while maintaining correct sort order and minimizing the length of the string. NUMKEYS also provides functions to pack ASCII strings, creating keys that are as much as 50% shorter. It also provides a routine to invert a string, so that an index will sort in descending order. See Section 6.G for more information.

Example

See Sections 4.C and 4.F.

See Also

BTAddRec
BTSetSearchForSequential

BTDeleteKey

Declaration

```
procedure BTAddRec(IFBPtr : IsamFileBlockPtr;  
                  var RefNr : LongInt;  
                  var Source);
```

Purpose

Add a data record.

Description

A data record is added to the data file with <BTAddRec>. <Source> is written to the end of the data file if it has no deleted entries, otherwise a record previously deleted with <BTDeleteRec> is overwritten and reused. <RefNr> returns the data reference number for the record after a successful return from <BTAddRec>. This long integer is used to identify the record for further access. <RefNr> will usually be passed in the <UserDatRef> parameter to <BTAddKey> soon after the record is added.

Since <Source> is an untyped parameter, any desired data structure can be passed to <BTAddRec>. Although this is necessary to allow <BTAddRec> to work for all data types, it transfers the responsibility to the programmer to pass a variable of the correct type. <BTAddRec> will always write the number of bytes specified in <DatSLen> when <BTCreateFileBlock> was called.

If the first four byte field of the data record has been reserved for a deletion tag, be sure to initialize it to zero before calling <BTAddRec>.

<BTAddRec> can be used with a network fileblock only if the fileblock has been locked for exclusive write access (level 1 or 2).

Example

See Sections 4.C and 4.F.

See Also

BTAddKey
BTGetRec

BTDeleteRec

BTaRecIsLocked / BTClearKey

Declaration

```
function BTaRecIsLocked(IFBPtr : IsamFileBlockPtr) : Boolean;
```

Purpose

Returns True if any record has been locked.

Description

B-Tree Filer maintains an internal list of the records locked by each workstation. <BTaRecIsLocked> returns True if the current workstation has locked one or more records. Note that this routine does not indicate whether another workstation has locked any records.

See Also

BTLockRec

BTRecIsLocked

Declaration

```
procedure BTClearKey(IFBPtr : IsamFileBlockPtr; Key : Word);
```

Purpose

Reset the sequential pointer.

Description

<BTClearKey> resets the sequential pointer of the index with number <Key>. This allows <BTNextKey> to return the first (smallest) key, and <BTPrevKey> the last (largest) key.

Example

```
BTClearKey(PF, KeyNr);  
if not IsamOK then begin  
  {Error handling}  
end;  
BTNextKey(PF, KeyNr, RefNr, KeyStr);  
...
```

After calling <BTClearKey> a following call to <BTNextKey> returns the first key in index KeyNr.

See Also

BTNextKey

BTPrevKey

BTCloseAllFileBlocks

Declaration

procedure BTCloseAllFileBlocks;

Purpose

Closes all currently opened fileblocks.

Description

All open fileblocks are closed, and all memory used by those fileblocks is freed. More information can be found in the description of the procedure <BTCloseFileBlock>.

<BTCloseAllFileBlocks> continues to close fileblocks even if an error occurs. The value returned in <IsamError> is the first error encountered.

Unlike <BTCloseFileBlock>, <BTCloseAllFileBlocks> cannot set each user variable of type <IsamFileBlockPtr> to Nil when the fileblock is closed. If an application repeatedly opens fileblocks after closing them with <BTCloseAllFileBlocks> it is a good idea to set the fileblock variables to Nil after closing them.

Example

```
BTCloseAllFileBlocks;  
if not IsamOK then begin  
  {Error handling}  
end;  
BTExitIsam;  
if not IsamOK then begin  
  {Error handling}  
end;
```

All fileblocks are closed before deinstalling B-Tree Filer.

See Also

BTCloseFileBlock

BTCloseFileBlock

Declaration

```
procedure BTCloseFileBlock(var IFBPtr : IsamFileBlockPtr);
```

Purpose

Close the specified fileblock.

Description

The fileblock is closed and the memory allocated by <BTOpenFileBlock> is freed. <IFBPtr> is set to Nil after a successful call to <BTCloseFileBlock>.

All locks--write locks, read locks, record locks--placed by the calling workstation are automatically removed. Error 10322 indicates that an attempt to remove an existing fileblock lock or readlock failed. Error 10323 indicates that an attempt to remove an existing record lock failed.

Any data or index information that remains in B-Tree Filer or DOS memory buffers is flushed to disk during this call. An appropriate error code is returned if the flush fails. Only after a successful flush are the files that correspond to the fileblock closed. If an error occurs while closing a file, error code 10160 is returned. Even so, memory used by the fileblock is freed and <BTCloseFileBlock> may not be called again for this fileblock.

The FILER unit implements a Turbo Pascal exit procedure that automatically closes all open fileblocks when the program halts, either successfully or with a runtime error. However, if a program crashes while fileblocks are still open and index writes have been performed, or if the call to <BTCloseFileBlock> fails within the exit procedure, it is likely that the index file will need to be rebuilt before the fileblock can be successfully opened again.

Example

See Section 4.C.

See Also

BTCloseAllFileBlocks
BTOpenFileBlock

BTCreatFileBlock

Declaration

```
procedure BTCreatFileBlock(FName : IsamFileBlockName;  
                           DatSlen : LongInt;  
                           NumberOfKeys : Word;  
                           IID : IsamIndDescr);
```

Purpose

Create a new fileblock with name <FName>.

Description

<BTCreatFileBlock> creates new data and index files based on the name <FName>. <FName> must contain either one, two, or three valid DOS filenames, separated by semicolons, with optional drive and pathname specifications. No extensions should be specified since these are automatically appended using <DatExtension> (default='DAT') for the data file and <IxExtension> (default='IX') for the index file. See the discussion of type <IsamFileBlockName> near the beginning of this chapter for more information about how to specify <FName>.

Existing files of the same name are overwritten without warning (although this may fail if another workstation has such files open in a non-shareable mode).

The parameter <DatSlen> specifies the length of each data record. (For variable length records, <DatSlen> specifies the section length. See Section 6.B for more information.) <DatSlen> must be in the range 21 to MaxLongInt bytes (although the practical upper limit is 65520 bytes). Error 10020 indicates an invalid record length.

In most cases, a data record definition should begin with a four byte (LongInt) field reserved for a deleted record tag. This field will contain four zero bytes for valid records, and a non-zero value for deleted records. Several B-Tree Filer units require this field: automatic fileblock rebuilding (REBUILD, REORG), variable length records (VREC, VREBUILD, VREORG), and fileblock browsing (BROWSER).

A data record is handled as a typeless parameter. B-Tree Filer maintains no internal knowledge of the fields within each record, only its length.

<NumberOfKeys> specifies the number of indexes for this fileblock. Valid values range from 0 to <MaxNrOfKeys>, inclusive. If <NumberOfKeys> is 0, no index file is created.

The parameter <IID> determines the length of each key and whether its associated index allows duplicate key strings. Typically, the <IID> parameter should be declared as a local variable and initialized just prior to calling

<BTCreatFileBlock>. For more details, see the programming example in Section 4.C and the description of type <IsamIndDescr> near the beginning of this chapter.

BTCreatFileBlock

<BTCreatFileBlock> temporarily allocates heap space (up to about 500 bytes) for internal data structures. It will return error 10030 or 10090 if insufficient memory is available.

The resulting data file is one record long. The first 21 bytes of this record contain B-Tree Filer system information matching the following template:

```
SystemRecord =  
  record  
    FirstFree : LongInt; {Ref number of first deleted record, -1 if none}  
    NumberFree : LongInt; {Number of deleted records}  
    NumRec : LongInt; {Total number of records}  
    LenRec : LongInt; {Bytes in one record}  
    NrOfKeys : LongInt; {Number of indexes for the fileblock}  
    Changed : Boolean; {True if index or data changed but not flushed}  
  end;
```

An application should not modify the contents of this system record.

The resulting index file contains space for <PageSize> keys of each index. The corresponding number of bytes can be obtained by calling <BTKeyRecordSize> after the fileblock has been opened. Don't be surprised if you have an index file of several thousand bytes even though you haven't added any keys yet.

Note that <BTCreatFileBlock> does not initialize a variable of type <IsamFileBlockPtr> and that the files it creates are left closed after the call. To use the fileblock, you must subsequently call <BTOpenFileBlock>.

Example

See Section 4.C.

See Also

BTDeleteFileBlock

BTOpenFileBlock

BTDataFileName / BTDatRecordSize

Declaration

```
function BTDataFileName(IFBPtr : IsamFileBlockPtr) : IsamFileName;
```

Purpose

Return the name of the fileblock's data file.

Description

This function returns the data file name, including extension, of the specified fileblock. If the name passed to <BTOpenFileBlock> did not include a drive and directory, neither does the string returned by <BTDataFileName>.

See Also

BTIndexFileName

Declaration

```
function BTDatRecordSize(IFBPtr : IsamFileBlockPtr) : LongInt;
```

Purpose

Return the size of a data record.

Description

Returns the value of the parameter <DatSLen> that was passed to <BTCreateFileBlock>.

If <BTFreeRecs> returns zero, <BTDatRecordSize> returns the number of bytes of disk space that <BTAddRec> will allocate.

Example

```
if BTFreeRecs(PF) = 0 then
  if IsamOK then
    Writeln('The next call to <BTAddRec> will increase data file size by '
      BTDatRecordSize(PF), ' bytes')
  else begin
    {Error handling}
  end;
```

Displays the increase in data file size when a record is added only if no free record holes exist.

See Also

BTKeyRecordSize

BTMinimumDatKeys

BTDeleteAllKeys

Declaration

```
procedure BTDeleteAllKeys(IFBPtr : IsamFileBlockPtr; Key : Word);
```

Purpose

Delete all keys in index <Key>.

Description

This procedure provides support for temporary keys. If a program needs to create a new index for a specific key number at runtime, this key can then quickly and easily be deleted with <BTDeleteAllKeys>. Note that the index "slot" for the temporary key must be reserved when the fileblock is first created. (See Section 6.C for an alternate approach.)

<BTDeleteAllKeys> marks the associated index space available for reuse but does not actually write over the existing keys.

<BTDeleteAllKeys> can be used with a network fileblock only if the fileblock has been locked for exclusive write access (level 1 or 2).

Example

In a fileblock with five indexes, assume that the fifth index (a secondary key) is used for generation of special lists and that when a data record is added, the fifth key is not added or is added with an empty string. When a special list needs to be created, <BTDeleteAllKeys> is used as follows:

```
var
  Ref      : LongInt;
  IKS      : IsamKeyStr;
  PTemp    : PersonDef;
...
  {Clear prior temporary keys}
  BTDeleteAllKeys(PF, 5);
  {Prepare to add new special keys}
  if IsamOK then BTClearKey(PF, 1);
  if IsamOK then BTNextKey(PF, 1, Ref, IKS);
  {Scan all records}
  while IsamOK do begin
    BTGetRec(PF, Ref, PTemp, False);
    {Add special keys}
    if IsamOK then BAddKey(PF, 5, Ref, BuildSpecKey(PTemp));
    if IsamOK then BTNextKey(PF, 1, Ref, IKS);
  end;
  if IsamError <> 10250 then
    {Error handling} ;
...

```

All data records are scanned using the first key and a special key is created and entered for each one. If the while loop ends with any error but 10250 (no larger key found), a real error must have occurred.

See Also

BTDeleteKey

Declaration

```
procedure BTDeleteFileBlock(FName : IsamFileBlockName);
```

Purpose

Delete a fileblock.

Description

Deletes all DOS files that correspond to the fileblock of name <FName>. The parameter <FName> may contain up to three semicolon-separated names, as is described by the type <IsamFileBlockName>. The third file name is always ignored by <BTDeleteFileBlock>.

The data ('DAT') and index ('IX') files of the fileblock are deleted, as well as the dialog ('DIA') file if it exists. Files with extensions 'MSG' and 'SAV', created by the procedure <RebuildFileBlock> among others, are never deleted.

Example

```
BTDeleteFileBlock( 'C:\DATA\STUFF;D:\INDEX\STUFF');  
if not IsamOK then begin  
  (Error handling)  
end;
```

The following files are deleted:

```
'C:\DATA\STUFF.DAT'  
'D:\INDEX\STUFF.IX'  
'C:\DATA\STUFF.DIA' (if any)
```

See Also

BTCreateFileBlock

BTDeleteKey

Declaration

```
procedure BTDeleteKey(IFBPtr : IsamFileBlockPtr; Key : Word;  
                     UserDatRef : LongInt; UserKey : IsamKeyStr);
```

Purpose

Delete a key.

Description

When a data record is deleted with <BTDeleteRec>, the index file also needs to be updated. This occurs by calling <BTDeleteKey> for every index in the fileblock. <Key> is the index number. <UserKey> is the key string and <UserDatRef> the associated data reference number, which is usually the parameter returned by one of the key search operations (e.g. <BTFindKey>). <UserDatRef> is ignored for a primary index, but must contain the correct value to delete a secondary key.

Error 10220 means that the key <UserKey> (in combination with <UserDatRef> for secondary indexes) was not found and therefore could not be deleted.

<BTDeleteKey> modifies the internal sequential pointer whether or not the <BTDeleteKey> call succeeds. Hence, the sequential pointer must be refreshed before calling <BTNextKey> or <BTPrevKey>. Although the automatic sequential searching option activated by <BTSetSearchForSequential> can sometimes recover from the modification, it is usually necessary to take additional precautions when a key is deleted. See the example below.

<BTDeleteKey> can be used with a network fileblock only if the fileblock has been locked for exclusive write access (level 1 or 2).

Example

```
var  
  Ref, NextRef : LongInt;  
  IKS, NextIKS : IsamKeyStr;  
  OK           : Boolean;  
...  
BTClearKey(PF, 1);  
if IsamOK then BTNextKey(PF, 1, Ref, IKS);  
while IsamOK do begin  
  if WantToDeleteThisRecord then begin  
    (Find the next key now)  
    BTNextKey(PF, 1, NextRef, NextIKS);  
    OK := IsamOK;  
    (Delete record and associated keys)  
    BTDeleteKey(PF, 1, Ref, IKS);  
    if IsamOK then BTDeleteRec(PF, Ref);  
    (Reinitialize the sequential pointer)  
    if OK then BTFindKeyAndRef(PF, 1, NextRef, NextIKS);  
    IsamOK := IsamOK and OK;  
  end else  
    BTNextKey(PF, 1, Ref, IKS);  
end;
```

BTDeleteKey

This example scans all keys in index 1 (assumed to allow duplicates) and uses some criterion to decide whether to delete each associated record. If a record and its keys are to be deleted, a call to <BTNextKey> determines the next reference number and key string prior to deleting the key. After the record and key are deleted, a call to <BTFindKeyAndRef> repositions the sequential pointer over the next key.

See Sections 4.C and 4.F for additional examples.

See Also

BTAddKey

BTDeleteRec

Declaration

procedure BTDeleteRec(IFBPtr : IsamFileBlockPtr; RefNr : LongInt);

Purpose

Delete a data record.

Description

<BTDeleteRec> deletes a data record from the data file. The record is really just marked as free, so that a future <BTAddRec> can fill this "hole".

Only existing (non-deleted) data records may be deleted. Deleting an already deleted data record corrupts an internally maintained linked list of deleted records and therefore has unpredictable results. When a record is deleted, B-Tree Filer sets the first four bytes of the record to a non-zero value. If the first four bytes of each data record are reserved for this purpose and normally set to zero, deleted records may be recognized easily.

An attempt to delete record 0 will generate error 10135. Record 0 contains internal system information and is never used as a normal data record.

<BTDeleteRec> can be used with a network fileblock only if the fileblock has been locked for exclusive write access (level 1 or 2).

Example

See Sections 4.C and 4.F.

See Also

BTAddRec

BTDeleteKey

BTDestroyBufferContents

Declaration

```
procedure BTDestroyBufferContents(IFBPtr : IsamFileBlockPtr;  
                                IgnoredUpdated : Boolean);
```

Purpose

Marks the contents of all index buffers of fileblock <IFBPtr> as invalid.

Description

In order to speed access to index information, B-Tree Filer normally buffers parts of the index file in RAM memory. Normally there is no reason for an application program to invalidate these buffers, since B-Tree Filer automatically does so when needed, e.g., in a multi-user situation. Invalidating the buffers of a given file requires more time for the next operation, since the index information must be read from disk again.

In special circumstances it may be necessary for an application programmer to invalidate the buffers, however. The particular case for which this routine was added arises with Novell's Transaction Tracking System (TTS). If a transaction is aborted, NetWare automatically backs out any changes to disk files that were made since the start of the transaction. Updated information may remain in B-Tree Filer's index buffers in memory, however. In this case, <BTDestroyBufferContents> should be called for any fileblocks that were involved in the transaction.

In the case just described, the <IgnoreUpdated> parameter should be set to True. This means that FILER will invalidate the buffers even though they would normally need to be flushed to disk. If you set <IgnoreUpdated> to False, and modified buffers are found, <BTDestroyBufferContents> will exit with <IsamError> 10447 without invalidating any modified buffers.

See Also

TTSAbort (NETWARE)

BTExitIsam

Declaration

procedure BTExitIsam;

Purpose

Finish working with B-Tree Filer.

Description

This procedure frees the memory allocated by <BTInitIsam> on the normal heap and in EMS memory. After calling <BTExitIsam>, all B-Tree Filer routines are prevented from working. (All calls will generate error 10455.) Only another call to <BTInitIsam> will allow further work with B-Tree Filer.

All fileblocks should be closed before calling <BTExitIsam>. If fileblocks remain open, <BTExitIsam> will attempt to flush data and close the fileblocks before freeing the memory buffers. If an error occurs while closing a fileblock, <BTExitIsam> attempts to close the remaining open fileblocks. The returned error code does not provide any information as to which fileblock could not be closed. However, in every case the error code pertains to the first error encountered. It is important to check the variable <IsamError> upon return from <BTExitIsam>.

Example

```
var
  PBS : LongInt;
...
PBS := BTInitIsam(NoNet, 50000, 0);
if not IsamOK then begin
  {Error handling}
end;
...
BTExitIsam;
if not IsamOK then begin
  {Error handling}
end;
...
PBS := BTInitIsam(Novell, 50000+MinimizeUseOfNormalHeap, 120);
if not IsamOK then begin
  {Error handling}
end;
...

```

B-Tree Filer is first initialized with <NoNet>. The buffers use 50000 bytes from the heap; the EMS heap is not used. After some work, B-Tree Filer is exited with a call to <BTExitIsam> and the buffers are freed. After additional non-Filer activity, B-Tree Filer is initialized again. This time it is initialized for the Novell network. The normal heap is used to store required pages only if there are not enough pages for the buffers in the EMS Heap.

See Also

BTInitIsam

Declaration

```
function BTFFileBlockIsLocked(IFBPTr : IsamFileBlockPtr) : Boolean;
```

Purpose

Check whether the fileblock is locked.

Description

This function determines whether the fileblock has been locked (for writing) by the calling workstation. If so, True is returned, otherwise False.

This function will always return False if B-Tree Filer was compiled with the compiler directive NoNet or if <IFBPTr> is a single user fileblock. If NoNet wasn't defined, i.e., a real network interface was activated, the present lock state of a network fileblock is returned. This is also the case when a network emulation mode is active (see Section 4.G).

Note that <BTFFileBlockIsLocked> doesn't determine whether another workstation has locked the fileblock. That may be determined only by attempting to lock the fileblock and checking for success.

Example

```
var
  Locked : Boolean;
...
  Locked := BTFFileBlockIsLocked(PF);
  if not Locked then begin
    TryToLockFileBlock(PF);
    if not IsamOK then begin
      (Error handling)
    end;
  end;
...
  if not Locked then begin
    BTUnlockFileBlock(PF);
    if not IsamOK then begin
      (Error handling)
    end;
  end;
```

This example will only lock a fileblock if it wasn't already locked.

TryToLockFileBlock would be a procedure that would attempt retries to lock a fileblock. Later, the fileblock lock will be released only if there was no fileblock lock earlier. In this way the code remains "neutral" to the outside world with respect to the lock state of the fileblock.

See Also

BTFFileBlockIsReadLocked

BTFFileBlockIsOpen

Declaration

```
function BTFFileBlockIsOpen(IFBPtr : IsamFileBlockPtr) : Boolean;
```

Purpose

Check whether the fileblock is open.

Description

Returns True if the fileblock is open, False if not. This function allows an application to avoid opening a fileblock more than once simultaneously, which can cause odd DOS-specific side effects besides just wasting file handles and heap space.

If two or more variables of type <IsamFileBlockPtr> are being reused to represent several actual fileblocks, it is important for the application to assure that the application's <IsamFileBlockPtr> variable is set to Nil as soon as the fileblock is closed. Otherwise it is possible that <BTFFileBlockIsOpen> may return an incorrect value of True when it is passed a parameter containing a previously initialized value.

Example

```
var
  Open : Boolean;
...
Open := BTFFileBlockIsOpen(PF);
if not Open then begin
  BTOpenFileBlock(PF, FName, False, False, False, False);
  if not IsamOK then begin
    {Error handling}
  end;
end;
...
if not Open then begin
  BTCloseFileBlock ( PF );
  if not IsamOK then begin
    {Error handling}
  end;
end;
end;
```

Opens a fileblock only if it was not previously opened. Subsequently the fileblock is closed only if it was previously closed. In this way the code remains "neutral" to the outside world with respect to the open state of the fileblock.

See Also

BTCloseFileBlock

BTOpenFileBlock

BTFFileBlockIsReadLocked

Declaration

```
function BTFFileBlockIsReadLocked(IFBPtr : IsamFileBlockPtr) : Boolean;
```

Purpose

Check whether the fileblock is read locked.

Description

This function determines whether the fileblock has been locked (for reading) by the calling workstation. If so, True is returned, otherwise False.

This function will always return False if B-Tree Filer was compiled with the compiler directive NoNet or if <IFBPtr>^ is a single user fileblock. If NoNet wasn't defined, i.e., a real network interface was activated, the present read lock state of the network fileblock is returned. This is also the case when a network emulation mode is active (see Section 4.G).

Note that <BTFFileBlockIsReadLocked> doesn't determine whether another workstation has locked the fileblock. That may be determined only by attempting to lock the fileblock and checking for success.

Example

```
var
  ReadLocked : Boolean;
...
ReadLocked := BTFFileBlockIsReadLocked(PF);
if not ReadLocked then begin
  TryToReadLockFileBlock (PF);
  if not IsamOK then begin
    {Error handling}
  end;
end;
...
if not ReadLocked then begin
  BTUnlockFileBlock(PF);
  if not IsamOK then begin
    {Error handling}
  end;
end;
```

This example will only read lock a fileblock if it wasn't already read locked.

TryToReadLockFileBlock would be a procedure that would attempt retries to readlock a fileblock. Later, the fileblock readlock will be released only if there was no readlock earlier. In this way the code remains "neutral" to the outside world with respect to the lock state of the fileblock.

See Also

BTFFileBlockIsLocked

BTFileLen

Declaration

```
function BTFileLen(IFBPtr : IsamFileBlockPtr) : LongInt;
```

Purpose

Determine the total number of records in the data file.

Description

This record count includes the first reserved data record, the deleted records, and the presently used data records.

The physical length of the data file can be calculated as follows:

```
BTDatRecordSize(IFBPtr) * BTFileLen(IFBPtr)
```

Example

```
var
  FLen : LongInt;
...
  FLen := BTFileLen(PF);
  if not IsamOK then begin
    {Error handling}
  end;
  Writeln('The total number of records is ', FLen);
  Writeln('Data file size is ', BTDatRecordSize(PF)*FLen, ' bytes');
```

In a network environment <BTFileLen> must read the header of the fileblock, which could lead to an error. In contrast the value returned by <BTDatRecordSize> is obtained without accessing the disk and therefore can never encounter an error.

See Also

BTFreeRecs

BTUsedRecs

Declaration

```
procedure BTFindKey(IFBPtr : IsamFileBlockPtr; Key : Word;  
    var UserDatRef : LongInt; UserKey : IsamKeyStr);
```

Purpose

Search for a specific key.

Description

The index file is searched for an exact match with the key <UserKey> of index <Key>. The matching data record reference is returned in <UserDatRef>. <BTFindKey> sets the sequential pointer upon successful execution, so that the procedures <BTPrevKey> and <BTNextKey> may be used thereafter.

If a secondary key that is entered more than once in the index file is found, the smallest matching data record reference is returned in <UserDatRef>. Successive calls to <BTNextKey> return increasingly larger values for <UserDatRef> for the identical key. If no data records have ever been deleted, this order is the same one in which the records were added to the fileblock. Never depend on this order, however, since by deleting a data record and then adding a new one, this order may be changed.

If the specified key string is not found, the variable <IsamError> will contain the value 10200. If another workstation has the fileblock locked for writing, <IsamError> will contain 10399.

Example

See Sections 4.C and 4.F.

See Also

BTFindKeyAndRef
BTSearchKey

BTKeyExists
BTSearchKeyAndRef

BTFindKeyAndRef

Declaration

```
procedure BTFindKeyAndRef(IFBPtr : IsamFileBlockPtr; Key : Word;
    var UserDatRef : LongInt;
    var UserKey : IsamKeyStr;
    NotFoundSearchDirection : Integer);
```

Purpose

Search for a key with the given data record reference.

Description

This procedure searches for the key <UserKey> that also has the data reference number <UserDatRef>. If this combination is not found, the parameter <NotFoundSearchDirection> determines whether to search further. A value of 0 means to end the search. Any positive value means to search for a following key and every negative value searches for a previous key.

If no key with the given reference is found, the variable <IsamError> will contain the value 10270. If <NotFoundSearchDirection> doesn't have a value of 0, an error code associated with the procedures <BTNextKey> and <BTPrevKey> will be returned if the additional search is not successful.

If another workstation has the fileblock locked for writing, <IsamError> will contain 10399.

This procedure is not designed to return the data reference number of a specific key; this must already be known. More importantly, it allows the sequential pointer to be positioned in the middle of a block of identical secondary keys. This is especially useful when browsing or creating lists.

<BTFindKeyAndRef> sets the sequential pointer upon successful execution, so that the procedures <BTPrevKey> and <BTNextKey> may be used thereafter.

Example

```
var
    Ref      : LongInt;
    SRef     : LongInt;
    IKS      : IsamKeyStr;
    SIKS     : IsamKeyStr;
    PTemp    : PersonDef;
    Stop     : Boolean;
...
BTFindKey(PF, 2, Ref, IKS);
Stop := False;
while IsamOK and (Not Stop) do begin
    BGetRec(PF, Ref, PTemp, False);
    if IsamOK then begin
        if PTemp.Age >= 18 then begin
            SRef := Ref;
            SIKS := IKS;
            Stop := True;
        end else
```


BTFindKeyAndRef

```
    BTNextKey(PF, 2, Ref, IKS);
end;
if BTIsamErrorClass > 1 then begin
    {Error handling}
end;
...
BTFindKeyAndRef(PF, 2, SRef, SIKS, 0);
if not IsamOK then begin
    {Error handling}
end;
```

The beginning of this example searches for a specific key in index number 2. The while loop then finds the first data record that has an Age field greater than or equal to 18. After checking for errors, further new key search operations may be performed. Finally <BTFindKeyAndRef> is used to set the sequential pointer on the data record that was found in the while loop.

See Also

BTFindKey
BTSearchKey

BTKeyExists
BTSearchKeyAndRef

BTFlushAllFileBlocks

Declaration

procedure BTFlushAllFileBlocks;

Purpose

Store all new data of all open fileblocks.

Description

This procedure writes all buffered new data for all open fileblocks to disk. For network fileblocks, only those that are locked will be written. Fileblocks opened in save mode never have data buffered in memory, so no action occurs for those fileblocks.

Example

This procedure need be called only when an application wants to assure that all files are completely updated on disk. A short meaningful example is therefore difficult to construct. An appropriate situation might exist when all open fileblocks contain related information and a transaction affecting several of the fileblocks has just been completed.

See Also

BTFlushFileBlock

Declaration

```
procedure BTFlushFileBlock(IFBPtr : IsamFileBlockPtr);
```

Purpose

Flush all new data for the specified fileblock.

Description

A fileblock's data and index information may be buffered in B-Tree Filer page buffers, in disk buffers managed by DOS, in disk caches, or in a file server's memory. This procedure writes all new data in such buffers to the physical disk device and updates the file directory entries. <BTFlushFileBlock> has the same effect as the sequence <BTCloseFileBlock>, <BTOpenFileBlock> would have except that <BTFlushFileBlock> is more efficient.

Calling this procedure is pointless for fileblocks that were opened in save mode, since there is never any new data in memory. <BTFlushFileBlock> does nothing for network fileblocks that are not locked for writing at the time of the call. No error is generated in this case.

The flushing mechanism used by <BTFlushFileBlock> varies depending on the value of the typed constant <IsamFlushDOS33> and on whether the fileblock is a network fileblock. See the description of <IsamFlushDOS33> near the beginning of this chapter for more information.

If <BTFlushFileBlock> fails because it cannot obtain a free file handle, <IsamError> will return 10150.

See Also

BTFlushAllFileBlocks
BTForceWritingMark

BTForceNetBufferWriteThrough

BTForceNetBufferWriteThrough

Declaration

```
procedure BTForceNetBufferWriteThrough(DoIt : Boolean);
```

Purpose

Activate write-through for all save mode files in a network environment.

Description

This routine is relevant only when one or more network fileblocks are being used in save mode. The default value for write-through is False.

Write-through refers to the process of flushing data from the memory buffers of a network file server or a multi-user operating system monitor. Many multi-user systems and networks buffer data being written to files instead of performing the write to disk immediately. Although the operating system will correctly provide the most recent data values to all workstations that request them, this buffering can cause lost data in the event of a crash at the file server, even for fileblocks opened in save mode.

Save mode normally flushes all data from B-Tree Filer's internal buffers before any B-Tree Filer call returns. However, save mode does not guarantee that operating system buffers are flushed. After you activate write-through by calling <BTForceNetBufferWriteThrough> with the parameter True, all save mode routines also flush operating system buffers before returning.

Automatic flushing activated via <BTForceNetBufferWriteThrough> uses the same techniques as does <BTFlushFileBlock>. The flushing mechanism varies depending on the value of the typed constant <IsamFlushDOS33>. See the description of <IsamFlushDOS33> near the beginning of this chapter for more information.

See Also

BTFlushFileBlock

BTForceWritingMark

Declaration

```
procedure BTForceWritingMark(FFM : Boolean);
```

Purpose

Write-through of the modified mark is enabled with <FFM> = True.

Description

When any B-Tree Filer procedure modifies the data or index file, a flag is set in the system record (reference number 0) of the data file. This flag indicates that data is buffered, either in B-Tree Filer internal buffers or in operating system file buffers. The flag remains set until the fileblock is closed or a flush is executed by calling <BTFlushFileBlock> or by an automatic internal flush. If a later call to <BTOpenFileBlock> finds the flag still set in the data file, it will return with an error code of 10010, which means that the integrity of the index and data files is questionable.

So far so good; unfortunately the flag itself may remain in an operating system buffer, and therefore this check for integrity may not be effective. This finally brings us to the purpose of <BTForceWritingMark>. Calling <BTForceWritingMark> with <FFM> set to True enables a mode in which the writing mark is immediately flushed to disk after it is written.

A call to this procedure with the value True increases the integrity of your data. The default value is False since the execution speed of B-Tree Filer is otherwise reduced slightly. For the greatest possible guarantee of data integrity (and additional degradation in speed), open the fileblock in save mode and activate <BTForceNetBufferWriteThrough>.

Example

```
BTAddRec(PF, RefNr, PersonRec);
```

If the computer is rebooted unexpectedly after this call to <BTAddRec>, the probability is high that neither the data record nor the header of the data file made it to the disk. In this case, the data file's modified flag will not indicate that the file integrity is questionable even though the data file is incomplete. If <BTForceWritingMark> had been called earlier with the parameter True, the modified flag would be guaranteed to indicate poor file integrity after the same events.

See Also

BTFlushFileBlock

BTForceNetBufferWriteThrough

BTFreeRecs

Declaration

```
function BTFreeRecs(IFBPtr : IsamFileBlockPtr) : LongInt;
```

Purpose

Determine the number of free data records.

Description

A "hole" exists in the data file for every record that has been deleted using the procedure <BTDeleteRec>. The "holes" will be refilled after later calls to <BTAddRec> or after the fileblock is rebuilt with <RebuildFileBlock>. <BTFreeRecs> returns the current number of these deleted records.

Example

```
var
  FRecs : LongInt;

...
FRecs := BTFreeRecs(PF);
if not IsamOK then begin
  (Error handling)
end;
Writeln('The number of free data records is ', FRecs);
Writeln('This is equivalent to ', BTDatRecordSize(PF)*FRecs,
  ' free bytes in the data file.');
```

In order to determine the present count of records in a network environment <BTFileLen> must read the header of the data file, which could lead to an error. By contrast the value returned from <BTDatRecordSize> is obtained without file access and therefore can never encounter an error.

See Also

BTUsedRecs

BTFileLen

Declaration

```
procedure BTGetApprKeyAndRef(IFBPtr : IsamFileBlockPtr; Key : Word;  
    RelPos : Word; Scale : Word;  
    var UserKey : IsamKeyStr;  
    var UserDatRef : LongInt);
```

Purpose

Return approximate <UserKey> and <UserDatRef> corresponding to relative position <RelPos> in the range 0..
<Scale>.

Description

Given a relative position <RelPos> in the range 0..
<Scale>, <BTGetApprKeyAndRef> returns the existing key <UserKey> and <UserDatRef> whose position within the B-tree most closely approximates the specified position. This is provided to convert a slider position within a scroll bar to the corresponding key and record number for highlighting a record within a file browser. <Scale> should typically be the number of screen characters assigned to the scroll bar, minus 1, and <RelPos> should correspond to the slider position relative to the base location of the scroll bar.

<BTGetApprKeyAndRef> is *not* intended to return an exact key string for a <RelPos> in the range 0..
<BTUsedKeys>-1, as would be required to create a Turbo Professional or Object Professional pick list that directly displayed the elements of a fileblock.

This procedure is written to obtain sufficient speed for interactive operation; good performance is achieved by reading only a small part of the B-tree to perform the calculations. The drawback to this method is the inexactness of the result. Only a full traversal of the tree could return an exact result, and this would lead to unacceptable execution time.

Large calculation errors occur only when the parameter <Scale> is too large. Acceptable values for <Scale> depend on the instantaneous state of the B-tree and the value of the constant <PageSize> (default = 62). Results are good if <Scale> is no larger than about $0.5 * \text{<PageSize>}$ or $0.67 * \text{<PageSize>}$ at most. The procedure always attempts to choose the key in the middle of the range that corresponds to equivalent <RelPos>.

To obtain the most stable results, it's a good idea to pass the values returned by <BTGetApprKeyAndRef> to <BTGetApprRelPos> and then use the <RelPos> returned by that routine to call <BTGetApprKeyAndRef> once more. In this way the two routines tend to cancel out errors.

<BTGetApprKeyAndRef> will return error 10280 if the specified index contains no keys. It will return error 10420 if <Scale> = 0 or if <RelPos> is larger than <Scale>.

BTGetApprKeyAndRef

Example

Assume that <Scale> is set to 19 and that there are 400 keys (numbered 1 to 400) in a B-tree. The following table shows the approximate keys returned by <BTGetApprKeyAndRef>:

<RelPos>	<UserKey>
0	10
1	30
10	210
19	390

Note that the extremes of the <RelPos> range do not return the extremes of the key range. Instead the returned keys are in the center of the group of keys that evaluate to the same <RelPos>.

See Also

BTGetApprRelPos

Declaration

```
procedure BTGetApprRelPos(IFBPtr : IsamFileBlockPtr; Key : Word;  
    var RelPos : Word; Scale : Word;  
    UserKey : IsamKeyStr; UserDatRef : LongInt);
```

Purpose

Compute the relative position of <UserKey> and <UserDatRef> in the B-tree, return <RelPos> in the range 0..<Scale>.

Description

Given <UserKey> and <UserDatRef>, which specify a key within index <Key> of a fileblock, <BTGetApprRelPos> returns a value <RelPos> in the range 0..<Scale>, which corresponds to the relative position of the key within the index. This routine is provided to convert a key string to a slider position within a scroll bar. <Scale> should typically be the number of screen characters assigned to the scroll bar, minus 1.

See <BTGetApprKeyAndRef> for further information.

Example

Assume that <Scale> is set to 19 and that there are 400 keys (numbered 1 to 400) in a B-tree. The following table shows the ranges of keys that return equivalent values of <RelPos>.

<UserKey> range	<RelPos> returned
1-20	0
21-40	1
201-220	10
381-400	19

See Also

BTGetApprKeyAndRef

BTGetRec

Declaration

```
procedure BTGetRec(IFBPtr : IsamFileBlockPtr; RefNr : LongInt;  
    var Dest; ISOLock : Boolean);
```

Purpose

Read the data record with reference number <RefNr>.

Description

<RefNr> is usually obtained from a prior index operation. For example, the <UserDatRef> parameter returned by a successful call to <BTFindKey> could be passed to <BTGetRec>.

Since <Dest> is untyped, the compiler will not detect an invalid variable passed in this position. Assure that the variable passed is large enough to hold a complete data record.

The parameter <ISOLock> only has meaning for a network fileblock. With a value of True, <BTGetRec> will read the record even if the fileblock is locked for writing (level 1 or 2) by another workstation. With a value of False, the operation is aborted and returns error 10399 when such a lock is found on the fileblock. A record lock (level 3) causes <BTGetRec> to fail no matter what the state of <ISOLock>. The procedure <BTGetRecReadOnly> must be used to get around such a lock.

The parameter <ISOLock> should be set to True with extreme care. It should be clear to the user that the data record might have just been changed and therefore possibly doesn't contain the newest data.

Example

See Sections 4.C and 4.F.

See Also

BTGetRecReadOnly

Declaration

```
procedure BTGetRecordInfo(IFBPtr : IsamFileBlockPtr; Ref : LongInt;  
    var Start, Len : LongInt; var Handle : Word);
```

Purpose

Return information about a specific data record.

Description

By using the information returned by <BTGetRecordInfo> an application may perform additional forms of record locking not provided directly by B-Tree Filer. The three var parameters <Start>, <Len>, and <Handle> fully describe where data record <Ref> is to be found. <Handle> is the file handle of the data file, <Start> is the offset of the first byte of the data record, and <Len> is the length of the data record.

Example

```
var  
    DataHandle : Word;  
    RecRef, RecStart, RecLen : LongInt;  
...  
function NovellLockRecShare(Start, Len : LongInt;  
    Handle : Word) : Boolean;  
type  
    LoHi = record Lo, Hi : Word; end;  
var  
    IRR : Registers;  
begin  
    with IRR do begin  
        BX := Handle;  
        CX := LoHi(Start).Hi; DX := LoHi(Start).Lo;  
        SI := LoHi(Len).Hi; DI := LoHi(Len).Lo;  
        BP := $0000;  
        AX := $BC03;  
        MsDos(IRR);  
        NovellLockRecShare := (AL = 0);  
    end;  
end;  
...  
BTFindKey(PF, 1, RecRef, 'SMITH');  
if not IsamOK then begin  
    {Error handling}  
end;  
BTGetRecordInfo(PF, RecRef, RecStart, RecLen, DataHandle);  
if not NovellLockRecShare(RecStart, RecLen, DataHandle) then begin  
    {Error handling}  
end;  
...  
...
```

This example implements a function that locks a file with a "shareable lock" in the Novell network. The data record reference returned by <BTFindKey> is passed to <BTGetRecordInfo>. The parameters returned by <BTGetRecordInfo> are passed to the Novell locking routine.

BTGetRecReadOnly

Declaration

```
procedure BTGetRecReadOnly(IFBPtr : IsamFileBlockPtr; RefNr : LongInt;  
                           var Dest);
```

Purpose

Read the data record <RefNr> even if the record is locked.

Description

<RefNr> is usually obtained from a prior index operation. For example, the <UserDatRef> parameter returned by a successful call to <BTFindKey> could be passed to <BTGetRecReadOnly>.

Since <Dest> is untyped, the compiler will not detect an invalid variable passed in this position. Assume that the variable passed is large enough to hold a complete data record.

<BTGetRecReadOnly> reads a fileblock locked by another workstation without difficulty. To this extent, it works identically to <BTGetRec> with <ISOLock> set to True. In addition, however, <BTGetRecReadOnly> will read a record even with a record lock. Because the first four bytes of the data record cannot be read when it is locked, these bytes are not initialized by <BTGetRecReadOnly> (only if the record is locked). However, if these bytes have been reserved for B-Tree Filer's use as a deletion tag, no critical information will be lost. The remaining bytes of the record are filled in as usual. When the record is locked and the first four bytes are lost, <BTGetRecReadOnly> returns a warning code of 10205 in <IsamError>.

<BTGetRecReadOnly> should be used only with extreme care. It should be clear to the user that the data record might have just been changed and therefore possibly doesn't contain the newest data. Because the first four bytes of a locked record cannot be read, it's also possible that the application may be dealing with a just-deleted record that it can't detect.

See Also

BTGetRec

BTGetSearchForSequential

Declaration

```
procedure BTGetSearchForSequential(IFBPtr : IsamFileBlockPtr; Key : Word;  
    var SFS : Boolean);
```

Purpose

Return the "SearchForSequential" state for index <Key> in <SFS>.

Description

This routine may be used to save the current state of the "SearchForSequential" option for a particular index and fileblock, so that the state may be changed and later restored. See <BTSetSearchForSequential> for further information.

Example

```
var  
    SFS : Boolean;  
...  
    BTGetSearchForSequential(PF, 1, SFS);  
    BTSetSearchForSequential(PF, 1, True);  
...  
    BTSetSearchForSequential(PF, 1, SFS);
```

To perform a certain set of index scanning operations it is imperative that "SearchForSequential" be enabled for index number 1. In order to preserve the prior setting of "SearchForSequential", its value is saved and then restored at the end.

See Also

BTSetSearchForSequential

BTGetStartingLong

Declaration

```
procedure BTGetStartingLong(IFBPtr : IsamFileBlockPtr; RefNr : LongInt;  
    var Dest : LongInt);
```

Purpose

Reads the first four bytes of record <RefNr> into <Dest>.

Description

This routine provides a fast way to read just the first four bytes of a record. The first four bytes are generally used to determine whether the specified record has been deleted.

Example

```
var  
    DelFlag : LongInt;  
...  
    BTGetStartingLong(PF, RefNr, DelFlag);  
    if IsamOK and (DelFlag = 0) then begin  
        {Record is not deleted}  
    ...  
    end;
```

Checks to see whether a record has been deleted before performing an operation on it.

See Also

BTGetRec

BTIndexFileName

Declaration

```
function BTIndexFileName(IFBPtr : IsamFileBlockPtr) : IsamFileName;
```

Purpose

Return the name of the fileblock's index file.

Description

This function returns the index file name, including extension, of the specified fileblock. If the name passed to <BTOpenFileBlock> did not include a drive and directory, neither does the string returned by <BTIndexFileName>.

See Also

BTDataFileName

BTInitIsam

Declaration

```
function BTInitIsam(ExpectedNet : NetSupportType;  
    Free : LongInt;  
    NrOfEMSTreePages : Word) : LongInt;
```

Purpose

This procedure initializes B-Tree Filer and allocates memory from the normal heap and/or the EMS heap.

Description

This function must be called before any routines of the FILER unit may be called either directly or indirectly.

<BTInitIsam> performs two tasks. First, the network to be used is initialized. The chosen network is specified via the parameter <ExpectedNet>. In order to successfully initialize a network, the appropriate compiler directives must be specified in BTDEFINE.INC in order to link the corresponding network interface code into the executable (see Section 2.A).

If BTDEFINE.INC specifies the NoNet define, <BTInitIsam> ignores the <ExpectedNet> parameter and initializes the FILER unit for single-user operation.

If BTDEFINE.INC defines one or more network interface directives, <ExpectedNet> may take any of the corresponding values in <NetSupportType>, in addition to <NoNet>. If <ExpectedNet> is set to <NoNet>, B-Tree Filer enables its network emulation mode and no network is initialized. In this case B-Tree Filer locking and unlocking routines don't really do anything. See Section 4.G for additional information about network emulation.

To enable true network operation, <ExpectedNet> must be set to a value other than <NoNet>. In some cases the global variable <IsamWSNr> must be initialized by the application prior to calling <BTInitIsam>. See Section 2.F for more information. A successful call to <BTInitIsam> initializes procedure pointers for routines that perform network-specific operations for locking, unlocking, and lock detection.

Note that the parameter <ExpectedNet> allows one to choose a network at runtime, thus allowing one program to be used on different networks. The determination of network type can be obtained from a command line parameter (as in the example NETDEMO.PAS) or by another method (as in NETINFO.PAS). Changing networks, for example from <Novell> to <NoNet>, is even possible at runtime. To do so, all open fileblocks must be closed and the procedure <BTExitIsam> must be called, in order to leave the previous network.

The second task performed by <BTInitIsam> is to configure internal index buffers. The parameters <Free> and <NrOfEMSTreePages> control the configuration.

<Free> concerns itself with the normal heap and <NrOfEMSTreePages> with the EMS Heap.

The <Free> parameter specifies how many bytes of the normal heap to leave available after the call. This allows some memory to be reserved for other purposes. We'll discuss how to determine a good value for <Free> in a moment. If <BTInitIsam> should allocate the absolute minimum amount of memory required for B-Tree Filer functionality from the normal heap, the value of <MinimizeUseOfNormalHeap> can be added to the value of <Free>.

<NrOfEMSTreePages> determines the maximum number of index pages that should be allocated on the EMS heap. To use the EMS heap at all, the conditional UseEMSHep must be defined in BTDEFINE.INC. Otherwise, the <NrOfEMSTreePages> parameter is ignored. More information can be found in Section 2.B.

It is not an error if the number of pages specified by <NrOfEMSTreePages> is not available. <BTInitIsam> will then use whatever it can obtain. If <NrOfEMSTreePages> or the obtained number of pages is less than the constant <MaxHeight> (default value of 8), additional buffers are allocated from the normal heap.

For every page allocated from the EMS heap, 25 bytes from the normal heap are also allocated in order to install a descriptor for the EMS page. As long as the value of pages in the EMS is not higher than a few hundred, this allocated memory is not significant. On the other hand, if more than 1000 pages are used in the EMS heap, the descriptors will use over 25000 bytes of the normal heap. Hence, the value of <Free> is taken into account even when allocating pages on the EMS Heap, in order to prevent the descriptors from using normal heap space required by another portion of the program.

The index buffers are organized as a circular list of descriptors, each of which contains a pointer to the actual index page. The descriptors are always allocated from the normal heap and each consume 25 bytes. The actual index pages are allocated either from the normal heap or in EMS memory. The index page size may be calculated as:

$$(\text{<MaxKeyLen>}+9) * \text{<PageSize>} + 18$$

Using the standard values for <MaxKeyLen> (30) and <PageSize> (62) leads to a memory requirement of 2436 bytes for each page.

At the minimum, <BTInitIsam> must be able to allocate <MaxHeight> page buffers. For the standard value of <MaxHeight> (8), the normal heap usage can thus range from a minimum of 200 bytes if all pages are stored in EMS, to 19688 bytes if all pages are stored on the normal heap.

BTInitIsam

<MaxKeyLen>, <MaxHeight>, and <PageSize> may be adjusted with caution. See the description of these constants near the beginning of this chapter.

The function return value of <BTInitIsam> is of type LongInt. If <IsamOK> is True after <BTInitIsam> returns, the low word of this LongInt contains the number of pages that were allocated in the normal heap, and the high word the number of pages in the EMS heap. If <IsamOK> is False and <IsamError> is 10000, the LongInt returns the number of index pages that could have been allocated. The difference between <MaxHeight> and this number may be used to calculate how much more memory is needed for a successful initialization. For any other <IsamError>, the LongInt function result returns 0.

So, how does one choose an appropriate value for <Free>? The answer depends on how the application uses the heap. Even if the program does not directly allocate heap space, other routines from B-Tree Filer require heap space themselves. For example, a call to <BTOpenFileBlock> uses about 500 bytes of heap space. Many windowing routines (such as those from Object Professional) also make liberal use of the heap.

If the amount of heap space needed for the rest of the program is known in advance, then the <Free> parameter should simply specify that amount of heap space plus a safety margin. For example, if a program will open two fileblocks and use no other heap space, then a safe value for <Free> would be 2000 (two 500 byte blocks plus 1000 bytes of margin). This value of <Free> would maximize the number of page buffers and would optimize indexing performance, especially in a single-user environment. (In a multi-user environment, page buffers are invalidated and must be reloaded whenever another workstation modifies the index of a fileblock. As a result, the efficacy of a large index buffer is not so clear when running on a network.)

At the other extreme is an application that makes extensive use of the heap itself; for example, a text editing program that stores text strings on the heap. In this case, you should use the special constant <MinimizeUseOfNormalHeap> to tell <BTInitIsam> to use as little normal heap space as possible. If no EMS memory is available, the minimum normal heap usage is 19688 bytes as calculated above for default settings. If sufficient EMS is available, normal heap usage can be limited to 225 bytes. See the examples below.

Important note: the minimum requirement for index buffers does not increase as more fileblocks are opened. B-Tree Filer shares the index buffers among all open fileblocks on a demand basis. However, performance may be improved by allocating an increased number of page buffers when many fileblocks are to be open simultaneously.

It is an error (10450) to call <BTInitIsam> more than once without an intervening call to <BTExitIsam>.

Examples

`Pages := BTInitIsam(Novell, 30000, 300);`

Initializes B-Tree Filer for Novell Netware. At least 30000 bytes of the normal heap remain free. Up to 300 page buffers are used on the EMS heap.

`Pages := BTInitIsam(Novell, 30000+MinimizeUseOfNormalHeap, 300);`

Initializes B-Tree Filer for Novell Netware. At least 30000 bytes of the normal heap remain free. Up to 300 buffers are used on the EMS heap. Only if less than `<MaxHeight>` buffers could be allocated from the EMS heap is memory allocated from the normal heap. For example, if only 3 buffers could be allocated in EMS, `<MaxHeight>-3 = 5` buffers would be allocated from the normal heap, subject to leaving at least 30000 bytes of normal heap space free.

`Pages := BTInitIsam(NoNet, 10000+MinimizeUseOfNormalHeap, 0);`

Initialize B-Tree Filer without a network. Keep at least 10000 bytes of the normal heap free. No buffers are allocated from the EMS heap, and `<MaxHeight>` buffers are allocated from the normal heap.

`Pages := BTInitIsam(MsNet, MinimizeUseOfNormalHeap, 0);`

Initializes B-Tree Filer for an MsNet-compatible network. Memory usage in normal RAM is minimized, and no EMS space is used.

BTIsamErrorClass

Declaration

function BTIsamErrorClass : Integer;

Purpose

Divide <IsamError> into one of five error classes.

Description

The function returns one of five error class values dependent upon <IsamError>. This provides a coarser and simpler picture of error recognition.

All error codes are summarized with their error classes in Appendix A. The following error classes can occur:

- 0 No error
- 1 User error
This is more of a warning than an error. The previously performed operation could not be successfully completed. Examples include a key that couldn't be found with <BTFindKey> and a record whose first four bytes couldn't be read with <BTGetRecReadOnly>. All values of the variable <IsamError> between 10200 and 10299 are user errors, among others.
- 2 Locking error (only on a network)
Errors of this class always occur when a file access couldn't be completed without some obvious reason for its failure (e.g., an invalid DOS file handle or a non-existent file). As a rule this error class occurs when a lock from another workstation prevents access. Also possible is an empty diskette drive or lack of privileges of the current user in the network. If after a retry the access is still not possible, the user should be given the opportunity to abort the operation.
- 3 Operation did not succeed (only in save mode)
This error class deals with serious errors and occurs only in save mode. All files of the corresponding fileblock are still functional but the desired operation could not succeed. Therefore this error class is a precursor to an error of class 4. If the condition that lead to the error cannot be corrected immediately, the best program response is to close up gracefully and abort.
- 4 Hard error
An immediate correction of the error condition is required. If this is not possible the program should abort. In some cases fileblocks will have been corrupted and require reconstruction.

Declaration

```
function BTIsNetFileBlock(IFBPtr : IsamFileBlockPtr) : Boolean;
```

Purpose

Returns True for a network fileblock.

Description

The term "network fileblock" is somewhat nebulous due to the numerous ways of configuring B-Tree Filer and opening fileblocks.

<BTIsNetFileBlock> always returns False when the FILER unit was compiled with the NoNet compiler directive. Since the code for use on a network was never compiled in, all fileblocks are automatically local. <BTIsNetFileBlock> also returns False whenever the parameter <Net> passed to <BTOpenFileBlock> was False when the fileblock was opened.

<BTIsNetFileBlock> will return True if the FILER unit was compiled with any directive besides NoNet and the <BTOpenFileBlock> was called with the <Net> parameter set to True. This occurs even if <BTInitIsam> was called with <ExpectedNet> set to <NoNet> because in this case B-Tree Filer emulates the network operation.

Example

```
{Make open request for a network fileblock}
BTOpenFileBlock(PF, FName, False, False, False, True);
if not BTIsNetFileBlock(PF) then begin
    ...
end;
```

There are two reasons why <BTIsNetFileBlock> might return False for this fileblock: either the FILER unit was compiled with the NoNet define in BTDEFINE.INC, or the value for <ExpectedNet> passed to <BTInitIsam> did not correspond to a network interface activated in BTDEFINE.INC.

See Also

BTNetSupported

BTNoNetCompiled

BTKeyExists

Declaration

```
function BTKeyExists(IFBPtr : IsamFileBlockPtr; Key : Word;  
                    UserDatRef : LongInt; UserKey : IsamKeyStr) : Boolean;
```

Purpose

Determines the existence of a particular key.

Description

Search for the key <UserKey> of index number <Key>. True is returned if the combination of <UserKey> and <UserDatRef> is found in the index, otherwise False.

If you are searching a primary index, the value of <UserDatRef> is never used by <BTKeyExists>.

If you are searching a secondary index and know the data reference number of the desired key, specify its value in the <UserDatRef> parameter. If you don't know the data reference number, specify zero for <UserDatRef>. In this case, <BTKeyExists> will return True if any key with the value <UserKey>, regardless of data reference number, exists in the index.

Note that this function is not for obtaining the data reference of a key; use <BTFindKey> for that purpose. The advantage of <BTKeyExists> is that it does not disturb the internal sequential pointer used by <BTNextKey> and <BTPrevKey>. For this reason and for its slightly faster speed <BTKeyExists> may be the preferred routine for certain applications.

Example

Suppose that one wishes to find all people in zip code 95060 (Santa Cruz) whose last name doesn't also occur in the zip code area 95066 (Scotts Valley). <BTKeyExists> can be used to formulate a solution as follows:

```
var  
    Person : PersonDef;  
    Name,  
    KeyStr1,  
    KeyStr2 : IsamKeyStr;  
    Ref1,  
    Ref2 : LongInt;  
    Match : Boolean;  
  
...  
KeyStr1 := '95060';  
BTSearchKey(PF, 2, Ref1, KeyStr1); {Index 2 is zip code}  
while IsamOK and (KeyStr1 = '95060') do begin  
    BTGetRec(PF, Ref1, Person, False);  
    if IsamOK then begin  
        Name := Copy(CreateKey(Person, 1), 1, 20);  
        KeyStr2 := Name;  
        BTSearchKey(PF, 1, Ref2, KeyStr2); {Index 1 is last name}  
    end;  
    Match := False;
```

```
while IsamOK and not Match and (Name = Copy(KeyStr2, 1, 20)) do begin
    Match := BTKeyExists(PF, 2, Ref2, '95066');
    if IsamOK and not Match then
        BTNextKey(PF, 1, Ref2, KeyStr2);
    end;
    If BTIsamErrorClass < 2 then begin
        IsamClearOK;
        if not Match then begin
            {Print out the data record}
        end;
    end;
    if IsamOK then
        BTNextKey(PF, 2, Ref1, KeyStr1);
end;
if BTIsamErrorClass > 1 then begin
    {Error handling}
end;
```

This example consists of two embedded while loops. The outer loop sequentially scans all records in zip code 95060. The inner loop obtains all data record references that correspond to the people who have the same last name. If one of these people lives in zip code 95066, the previously found person is not printed out.

At the point where <BTKeyExists> is used, there are two other ways to obtain the same information. Both of them have drawbacks. The first alternative is to read the data record with reference Ref2 and check the ZipCode field for '95066'. This requires an additional record variable and an additional data file access. The additional file access is unlikely with <BTKeyExists> because the corresponding page of the index tree is probably already in a buffer.

The second possibility is to use the procedure <BTFindKeyAndRef>, which is just a little slower than <BTKeyExists>. This approach is undesirable because the sequential pointer of index number 2 is changed. Consequently, the call to <BTNextKey> in the outer while loop would return the wrong value. To compensate we would need to call <BTFindKeyAndRef> just prior to <BTNextKey> in order to restore the sequential pointer to its original position. This costs additional time.

See Also

BTFindKey
BTSearchKey

BTFindKeyAndRef
BTSearchKeyAndRef

BTKeyRecordSize

Declaration

```
function BTKeyRecordSize(IFBPtr : IsamFileBlockPtr) : LongInt;
```

Purpose

Determine the size of a page block.

Description

When adding a key, <BTAddKey> will eventually need to allocate a new page on the disk (when a previous index page is filled and must be split). The amount of additional disk space for one new index page is returned by <BTKeyRecordSize>. The same value also determines the minimum size for a newly created index file.

The value doesn't change once a fileblock has been opened. Therefore there is no need to determine this value more than once.

The value returned by BTKeyRecordSize is determined using the following calculation:

```
var
  K : Word;
  KRS : LongInt;
begin
  KRS := 0;
  for K := 1 to BTNrOfKeys(IFBPtr) do
    inc(KRS, LongInt(KeyLen[K]+9)*PageSize+6);
  BTKeyRecordSize := KRS;
end;
```

where KeyLen[K] is the maximum key length (as originally specified in the <IsamIndDescr> passed to <BTCreateFileBlock>) for index number K. For example, if a fileblock had 5 indexes, each accepting keys up to 19 characters long (plus length byte), and <PageSize> had its standard value of 62, <BTKeyRecordSize> would be 9070 bytes.

Note that this function may also be used to estimate the size of an index file. Assuming that one key string is added for each index for each record, and that each index page is 75% full, the expected size of the index file is:

```
IndexSize := (1.5*BTUsedRecs(IFBPtr)*BTKeyRecordSize(IFBPtr)) div PageSize;
```

Example

```
if DiskFree(1) < BTKeyRecordSize(PF) then
  Writeln('Addition of a key may not be possible');
BTAddKey(PF, 1, Ref, KeyStr);
```

In this example the destination diskette is checked before a key is added to see whether enough space is available for the worst case expansion of the index file. Note that <BTAddKey> and other B-Tree Filer routines will detect an error if space runs out anyway.

See Also

BTDatRecordSize

BTMinimumDatKeys

BTLockAllOpenFileBlocks

Declaration

```
procedure BTLockAllOpenFileBlocks;
```

Purpose

Reserve all open network fileblocks for exclusive write access.

Description

All open network fileblocks are locked (referred to as a lock of level 1). This or a call to <BTLockFileBlock> is a prerequisite for modifying a network fileblock. Following this call, no other workstation can read or write these fileblocks, except by calling <BTGetRecReadOnly> or by calling <BTGetRec> or <BTPutRec> with parameter <ISOLock> set to True.

Calling this routine more than once is not an error. The network fileblocks remain locked. All open network fileblocks are set to the unlocked state if <BTLockAllOpenFileBlocks> fails for any reason.

Example

```
BTLockAllOpenFileBlocks;  
if BTIsamErrorClass = 2 then begin  
  Writeln('Locking attempt failed.');
```

```
end;
```

An error class of 2 means that the fileblock lock couldn't be obtained, possibly because another workstation already has a lock on a file.

See Also

BTLockFileBlock

BTReadLockAllOpenFileBlocks

BTUnlockAllOpenFileBlocks

BTLockFileBlock

Declaration

```
procedure BTLockFileBlock(IFBPtr : IsamFileBlockPtr);
```

Purpose

Reserve a network fileblock for exclusive write access.

Description

The network fileblock is locked (referred to as a lock of level 2). This or a call to <BTLockAllOpenFileBlocks> is a prerequisite for modifying a network fileblock with any of the following routines:

- <BTAddKey>
- <BTAddRec>
- <BTDeleteKey>
- <BTDeleteRec>
- <BTPutRec>

Following this call, no other workstation can read or write the fileblock, except by calling <BTGetRecReadOnly> or by calling <BTGetRec> or <BTPutRec> with parameter <ISOLock> set to True.

Calling this routine more than once is not an error. The network fileblock remains locked.

A fileblock is locked for exclusive write access in two steps. First, a physical lock is placed on the first four bytes of the dialog file. If <BTLockFileBlock> is unable to obtain this lock, it will exit immediately with <IsamError> 10330. Failure to obtain this first lock usually means that another workstation has the fileblock locked for its own exclusive write access.

Second, a physical lock is placed over a subsequent segment of the dialog file whose extent is proportional to the maximum number of workstations on the network. If <BTLockFileBlock> fails to obtain this lock, it will retry automatically using the typed constants <IsamDelayBetwLocks> and <IsamRetriesForLock> to determine the retry behavior. If it still cannot obtain the lock, it will exit with error 10330. Temporary failure to obtain the second portion of the lock usually occurs when another workstation has a temporary read lock on the fileblock. (Temporary read locks are used frequently for B-Tree Filer internal operations.) Sustained failure to obtain the second portion means that the application has placed an explicit read lock and has not released it.

An application usually needs to code its own retry loop calling <BTLockFileBlock> for a certain number of times and prompting the user whether to continue or abort the operation if the lock cannot be obtained. See NETDEMO.PAS for a functional example.

Care must be taken if more than one fileblock needs to be locked simultaneously to perform a transaction, since it is easily possible for two workstations to enter a "fatal embrace". It's often better to call <BTLockAllOpenFileBlocks> in this case,

since that routine guarantees that all of the fileblocks are locked, or none of them. See Section 4.F for more information.

To achieve good multi-user performance it is essential to minimize the amount of time that fileblocks remain locked. Again, see Section 4.F.

Example

```
BTLockFileBlock(PF);  
if BTIsamErrorClass = 2 then begin  
  Writeln('Locking attempt failed.');
```

```
end;
```

An error code class of 2 means that the fileblock lock couldn't be obtained because possibly another workstation already has a lock on the file.

A more complete example can be found in Section 4.F.

See Also

BTLockAllOpenFileBlocks
BTUnlockFileBlock

BTReadLockFileBlock

BTLockRec

Declaration

```
procedure BTLockRec(IFBPtr : IsamFileBlockPtr; Ref : LongInt);
```

Purpose

Reserve the data record with reference <Ref> for exclusive access.

Description

The data record with reference <Ref> in the data file of the network fileblock is locked (referred to as a lock of level 3). Following this call, no other workstation can read or write the record, with the one exception of calling <BTGetRecReadOnly> to read all but the first four bytes of the record.

This procedure is provided primarily to allow writing an edited record back to the data file without placing a fileblock lock. To do so, call <BTLockRec> for the record to be modified, then call <BTPutRec> with the parameter <ISOLock> set to True. In this way, the record can be modified even if another workstation has a fileblock lock in force, with the guarantee that no other workstation is simultaneously modifying the same record. The entire fileblock must always be locked in order to modify the index file. Be aware that using record locks in this way complicates logic throughout the program.

B-Tree Filer maintains an internal list of locked records. As a result, <BTLockRec> must sometimes allocate a small block of heap space (21 bytes for the default settings.) It will return error 10337 if unable to do so.

Calling this routine multiple times for the same <Ref> is not an error. B-Tree Filer recognizes the duplicate locking attempt and does not call the operating system to place the lock again.

Example

See Section 4.F.

See Also

BTLockFileBlock
BTUnlockRec

BTReadLockFileBlock

Declaration

```
function BTMinimumDatKeys(IFBPtr : IsamFileBlockPtr;  
                          Space : LongInt ) : LongInt;
```

Purpose

Determine the maximum capacity of a drive.

Description

Using this function one can determine how many data records and keys can be written to a storage device with <Space> bytes left. The worst case scenario for filling of the index tree is used (i.e., index pages are assumed to be only half full). It is therefore possible that up to 70% more data records with their keys can be written.

<BTMinimumDatKeys> does not access the disk, so it can be expected not to fail.

If more data records and their keys are to be written than <BTMinimumDatKeys> returns, <BTDatRecordSize> and <BTKeyRecordSize> can be used to test if this may safely be done.

Although this function is primarily useful for diskette drives, it may also be useful for hard drives. For example, it can determine at the onset how many data records with keys will definitely fit on a 20MB hard drive.

Example

```
Writeln('On drive C: at least: ',  
       BTMinimumDatKeys(PF, DiskFree(3)),  
       ' data records with keys can be stored.');
```

See Also

BTDatRecordSize

BTKeyRecordSize

BTNetSupported

Declaration

```
function BTNetSupported : NetSupportType;
```

Purpose

Return the currently initialized network.

Description

This function allows one to determine which network was initialized through the function <BTInitIsam>. This is primarily useful for showing status.

Note: <BTNetSupported> will return <NoNet> when BTDEFINE.INC defines the compiler directive NoNet and also when the <ExpectedNet> parameter passed to <BTInitIsam> is equal to <NoNet>. Use the function <BTNoNetCompiled> to differentiate between these two cases.

<BTNetSupported> can also be used as a debugging tool. If the net passed to <BTInitIsam> is not returned by <BTNetSupported> this means the network wasn't successfully initialized. This can occur if the <IsamWSNr> returned by the network isn't valid for B-Tree Filer or if the setting in BTDEFINE.INC is incorrect.

Example

```
const
  NetSupportString : array[NetSupportType] of String[13] =
    ('NoNet', 'Novell', 'MsNet', 'MsNetMachName', 'CBISNet',
     'PCMS386', 'VinesNet', 'NTNXNet', 'DesqView');

***
  Writeln('The current network is ', NetSupportString[BTNetSupported]);
```

The return value of <BTNetSupported> can be used as an index into an array of strings.

See Also

BTInitIsam

BTNoNetCompiled

BTIsNetFileBlock

Declaration

```
procedure BTNextDiffKey(IFBPtr : IsamFileBlockPtr; Key : Word;  
    var UserDatRef : LongInt; var UserKey : IsamKeyStr);
```

Purpose

Search for the next larger key that differs from <UserKey>.

Description

<NextDiffKey> searches index <Key> of the index file for the next key after <UserKey> that differs from it. The data reference number is returned in <UserDatRef> and the differing key string is returned in <UserKey>. The smallest data reference <UserDatRef> is returned for a secondary key that occurs more than once in the index file.

This procedure is especially useful to jump over a number of identical secondary keys and to obtain the next key, without performing a sequence of <BTNextKey> calls.

Only if no such key exists (<UserKey> is larger than or equal to the largest key) does the search fail and then <IsamError> 10240 is returned.

<BTNextDiffKey> sets the sequential pointer upon successful completion, so that the procedures <BTNextKey> and <BTPrevKey> can be used immediately after.

Example

```
var  
    KeyStr : IsamKeyStr;  
    Ref    : LongInt;  
...  
    KeyStr := '';  
    repeat  
        BTNextDiffKey(PF, 2, Ref, KeyStr);  
        if IsamOK then  
            Writeln(KeyStr);  
    until not IsamOK;  
    if BTIsamErrorClass > 1 then begin  
        {Error handling}  
    end;
```

This small piece of code reports all unique keys that are available in index 2.

See Also

BTNextKey

BTPrevDiffKey

BTNextKey

Declaration

```
procedure BTNextKey(IFBPtr : IsamFileBlockPtr; Key : Word;  
    var UserDatRef : Longint; var UserKey : IsamKeyStr);
```

Purpose

Obtain the next key in ascending sequence.

Description

<BTNextKey> advances one key further in index <Key> of the fileblock's index file. The next key string is returned in <UserKey> along with its corresponding data reference in <UserDatRef>. Commonly, <BTFindKey> or <BTSearchKey> is used to find a certain key and then <NextKey> is called to find all keys greater than the first until some criterion is met or no more keys exist.

If there is no next key available, <IsamError> is set to 10250.

The functionality of <BTNextKey> is dependent upon the validity of the internal sequential pointer. If the sequential pointer is valid, the next key is returned independent of the initial values of the parameters <UserKey> and <UserDatRef>. In this case both of these var parameters can be passed to <BTNextKey> in an uninitialized state.

If the sequential pointer has been disturbed--by calling <BTAddKey> or <BTDeleteKey> or if another workstation has called the same routines--the action of <BTNextKey> is dependent on the state of the "SearchForSequential" option for index <Key> of the fileblock. If the option is disabled, <BTNextKey> will return immediately with <IsamError> 10255. However, if the option is enabled (as it is by default), the parameters are used to reestablish the sequential pointer by calling <BTFindKey>. Then the next key is determined as usual. Hence, when calling <BTNextKey> in a loop, the values of <UserKey> and <UserDatRef> returned by the previous iteration should be passed into <BTNextKey> to enable automatic repositioning.

The internal pointer is valid after calls to the following procedures: <BTClearKey>, <BTSearchKey>, <BTFindKey>, <BTNextDiffKey>, <BTPrevDiffKey>, <BTFindKeyAndRef> and <BTSearchKeyAndRef>.

Example

```
var  
    KeyStr : IsamKeyStr;  
    Ref : Longint;  
    Person : PersonDef;  
...  
    KeyStr := '96000';  
    BTSearchKey(PF, 2, Ref, KeyStr);  
    while IsamOK and (KeyStr <= '96999') do begin  
        BTGetRec(PF, Ref, Person);  
        if IsamOK then begin  
            (display information about the Person)
```



```
    BTNextKey(PF, 2, Ref, KeyStr);  
  end;  
end;  
if BTIsamErrorClass > 1 then begin  
  {Error handling}  
end;
```

This is a typical use of the procedure <BTNextKey>. All persons who have zip codes between 96000 and 96999 are displayed.

In a network fileblock for which the "SearchForSequential" option has been enabled, this example will automatically correct itself if another workstation adds or deletes a key during the while loop. That's because the Ref and KeyStr variables returned by one call to <BTNextKey> are passed into the next call.

See Also

BTDeleteKey

BTPrevKey

BTNoNetCompiled

Declaration

```
function BTNoNetCompiled : Boolean;
```

Purpose

Determine whether B-Tree Filer was compiled with the NoNet define.

Description

This function allows one to determine at runtime whether the FILER unit was compiled with the compiler directive NoNet (True) or not (False). In combination with the function <BTNetSupported> an application can thus determine its network capabilities.

Example

```
const
  NetSupportString : array[NetSupportType] of String[13] =
    ('NoNet', 'Novell', 'MsNet', 'MsNetMachName', 'CBISNet',
     'PCDOS386', 'VinesNet', 'NTNXNet', 'DesqView');
...
  Writeln('The current network is ', NetSupportString[BTNetSupported]);
  if BTNetSupported = NoNet then
    if BTNoNetCompiled then
      Writeln('FILER was compiled with the NoNet define')
    else
      Writeln('FILER was initialized for network emulation');
```

The example from <BTNetSupported> was expanded upon here, in order to differentiate the return value of <NoNet>.

See Also

BTInitIsam

BTIsNetFileBlock

BTNetSupported

Declaration

```
function BTNrOfKeys(IFBPtr : IsamFileBlockPtr) : Word;
```

Purpose

Returns the number of indexes in the fileblock.

Description

This function allows one to determine at runtime how many indexes were defined when the fileblock was created with <BTCreateFileBlock>.

This is generally useful for informational purposes, but also can be part of general-purpose shells around the B-Tree Filer routines.

Example

```
var
  NumberKeys,
  KeyNr      : Word;
...
NumberKeys := BTNrOfKeys(PF);
if NumberKeys > 0 then begin
  Writeln('Please choose index number to Browse from 1 to ', NumberKeys);
  Readln(KeyNr);
end else
  Writeln('No index is available to browse.');
```

This code fragment shows how <BTNrOfKeys> can be used to determine the maximum number of indexes and how to use it for a general purpose querying routine.

Also see the examples in Section 4.C.

See Also

BTCreateFileBlock

BTOpenFileBlock

Declaration

```
procedure BTOpenFileBlock(var IFBPtr : IsamFileBlockPtr;  
                          FName : IsamFileBlockName;  
                          ReadOnly, AllReadOnly, Save, Net : Boolean);
```

Purpose

Opens an existing fileblock in the specified mode and returns a pointer to the fileblock descriptor.

Description

All B-Tree Filer routines that deal with a fileblock can be called only after the fileblock has been opened with <BTOpenFileBlock>.

The parameter <FName> contains up to three semicolon separated DOS filenames without extensions. The third one is ignored since it has no meaning to <BTOpenFileBlock>. See the discussion of type <IsamFileBlockName> near the beginning of this chapter for more information about how to specify <FName>.

Four boolean parameters determine the mode with which the fileblock is opened. The fileblock is opened for reading only if <ReadOnly> has a value of True. All attempts to perform a write operation with the fileblock will result in a locking error. Retry attempts will never succeed in this case. If an application allows certain users write access and other users read access only, it must be carefully coded so that the read-only users are not confused by error messages that report locks.

With a value of True for <AllReadOnly> a special situation occurs that only makes sense for a network environment. This mode specifies that all workstations have only read access, that no workstation may modify the fileblock. In this way, B-Tree Filer's internal buffers are guaranteed to remain valid at all times. As a result, internal operations that check and maintain the validity of these buffers do not have to be executed, resulting in faster execution. It is essential for the application to ensure that if any workstation sets <AllReadOnly> to True, then all workstations set <AllReadOnly> to True. Any one workstation writing to the fileblock will result in very confused reader workstations. The <AllReadOnly> option is provided primarily to optimize access to a read only drive such as a CD-ROM and should only be set to True for similar situations.

When <Save> is set to True, the fileblock is opened in save mode. This activates a data integrity enhancement scheme that decreases the execution speed of all writing routines of B-Tree Filer. More information can be found in Sections 4.A and 4.G.

If <Net> is True, the fileblock is opened as a network fileblock. Please note the requirements listed at the beginning of this chapter for successfully opening a network fileblock.

These four boolean parameters are not totally independent. Therefore the following automatic corrections are made in the passed arguments:

If <AllReadOnly> is True, <ReadOnly> is also set to True. If <ReadOnly> is True, then <Save> is set to False (no write operations can be performed). <Net> is always set to False if the FILER unit was compiled with the compiler directive "NoNet".

<BTOpenFileBlock> allocates memory from the heap in order to store certain data. The amount of heap space can be calculated as follows:

$$80 * (<NumberOfKeys> + 1) + 227$$

This memory does not need to be one contiguous chunk. It is actually used in multiple small pieces. An additional 33 bytes are required if the fileblock was opened with the parameter <Net> set to True.

Examples

```
BTOpenFileBlock(PF, FName, False, False, False, False);
```

This fileblock is opened for reading and writing without a network. The save mode is disabled.

```
BTOpenFileBlock(PF, FName, False, False, True, True);
```

This fileblock is opened for reading and writing on a network. The save mode is enabled.

```
BTOpenFileBlock(PF, FName, True, False, False, True);
```

This fileblock is opened only for reading on a network. The save mode is disabled.

```
BTOpenFileBlock(PF, FName, True, True, False, True);
```

This fileblock is opened only for reading on a network. In addition all workstations on the network have only read access. The save mode is disabled.

```
BTOpenFileBlock(PF, FName, False, True, True, True);
```

This fileblock is opened only for reading in a network (automatic correction of <ReadOnly>). In addition all workstations in the network have only read access. The save mode is disabled (automatic correction of <Save>).

See Also

BTCloseFileBlock

BTOtherWSChangedKey

Declaration

```
function BTOtherWSChangedKey(IFBPtr : IsamFileBlockPtr;  
                             Key : Word) : Boolean;
```

Purpose

Returns True if it is definite that index <Key> was changed by another workstation.

Description

B-Tree Filer's BROWSER unit may call this function to determine whether another workstation has modified the browsing index of the fileblock. In this way it can automatically update the screen to show changes without wasting too much effort.

<BTOtherWSChangedKey> is a relatively fast routine: it places a temporary read lock and reads a short area in the dialog file from which it can determine whether a change has occurred. If <BTOtherWSChangedKey> is unable to obtain the read lock, or if an error occurs while reading the dialog file, it always returns False.

Once <BTOtherWSChangedKey> returns True, it will continue to return True until a B-Tree Filer routine is called that reloads the page index buffers from the disk file. Such routines include <FindKey>, <FindKeyAndRef>, and other key access routines.

Example

```
while not KeyPressed do begin  
  if BTOtherWSChangedKey(PF, 1) then begin  
    FastWrite('Changed', 25, 70, $70);  
    Delay(1000);  
    BTFindKeyAndRef(PF, 1, CurDatRef, CurUserKey, 1);  
  end else  
    FastWrite('      ', 25, 70, $70);  
end;
```

Maintains a simple status line on row 25 that displays a message for one second whenever another workstation modifies index 1. This routine would be called whenever an application waited for a keypress. CurDatRef and CurUserKey must contain values for a known record.

Declaration

```
procedure BTPrevDiffKey(IFBPtr : IsamFileBlockPtr; Key : Word;  
    var UserDatRef : LongInt; var UserKey : IsamKeyStr);
```

Purpose

Search for the next smaller key that differs from <UserKey>.

Description

<PrevDiffKey> searches index <Key> of the index file for the last key before <UserKey> that differs from it. The data reference number is returned in <UserDatRef> and the differing key string is returned in <UserKey>. The largest data reference <UserDatRef> is returned for a secondary key that occurs more than once in the index file.

This procedure is especially useful to jump over a number of identical secondary keys and to obtain the previous key, without performing a sequence of <BTPrevKey> calls.

Only if no such key exists (<UserKey> is smaller than or equal to the smallest key) does the search fail and then <IsamError> 10245 is returned.

<BTPrevDiffKey> sets the sequential pointer upon successful completion, so that the procedures <BTNextKey> and <BTPrevKey> can be used immediately after.

Example

```
var  
    KeyStr : IsamKeyStr;  
    Ref    : LongInt;  
...  
KeyStr := #255;  
repeat  
    BTPrevDiffKey(PF, 2, Ref, KeyStr);  
    if IsamOK then  
        Writeln(KeyStr);  
until not IsamOK;  
if BTIsamErrorClass > 1 then begin  
    {Error handling}  
end;
```

This small piece of code reports all unique keys that are available in index 2, in descending order.

See Also

BTNextDiffKey

BTPrevKey

BTPrevKey

Declaration

```
procedure BTPrevKey(IFBPtr : IsamFileBlockPtr; Key : Word;  
    var UserDatRef : LongInt; var UserKey : IsamKeyStr);
```

Purpose

Obtain the previous key.

Description

<BTNextKey> moves one key back in index <Key> of the fileblock's index file. The previous key string is returned in <UserKey> along with its corresponding data reference in <UserDatRef>. Commonly, <BTFindKey> or <BTSearchKey> is used to find a certain key and then <PrevKey> is called to find all keys less than the first until some criterion is met or no more keys exist.

If there is no previous key available, <IsamError> is set to 10260.

The functionality of <BTPrevKey> is dependent upon the validity of the internal sequential pointer. If the sequential pointer is valid, the next key is returned independent of the initial values of the parameters <UserKey> and <UserDatRef>. In this case both of these var parameters can be passed to <BTPrevKey> in an uninitialized state.

If the sequential pointer has been disturbed--by calling <BTAddKey> or <BTDeleteKey> or if another workstation has called the same routines--the action of <BTPrevKey> is dependent on the state of the "SearchForSequential" option for index <Key> of the fileblock. If the option is disabled, <BTPrevKey> will return immediately with <IsamError> 10265. However, if the option is enabled (as it is by default), the parameters are used to reestablish the sequential pointer by calling <BTFindKeyAndRef>. Then the previous key is determined as usual. Hence, when calling <BTPrevKey> in a loop, the values of <UserKey> and <UserDatRef> returned by the previous iteration should be passed into <BTPrevKey> to enable automatic repositioning.

The internal pointer is valid after calls to the following procedures: <BTClearKey>, <BTSearchKey>, <BTFindKey>, <BTNextDiffKey>, <BTPrevDiffKey>, <BTFindKeyAndRef> and <BTSearchKeyAndRef>.

Example

```
var  
    KeyStr : IsamKeyStr;  
    Ref    : LongInt;  
    Person : PersonDef;  
...  
    KeyStr := '96000';  
    BTSearchKey(PF, 2, Ref, KeyStr);  
    while IsamOK and (KeyStr >= '95000') do begin  
        BGetRec(PF, Ref, Person);  
        if IsamOK then begin  
            {Display information about Person}
```



```
    BTPrevKey(PF, 2, Ref, KeyStr);  
  end;  
end;  
if BTIsamErrorClass > 1 then begin  
  (Error handling)  
end;
```

This is a typical use of the procedure <BTPrevKey>. All persons who have zip codes between 96000 and 95000 are displayed in descending order.

In a network fileblock for which the "SearchForSequential" option has been enabled, this example will automatically correct itself if another workstation adds or deletes a key during the while loop. That's because the Ref and KeyStr variables returned by one call to <BTPrevKey> are passed into the next call.

BTPutRec

Declaration

```
procedure BTPutRec(IFBPtr : IsamFileBlockPtr; RefNr : LongInt;  
    var Source; ISOLock : Boolean );
```

Purpose

Writes a data record back to the location <RefNr>.

Description

<RefNr> is usually retained from a previous <BTGetRec> operation. <BTPutRec> is used to write an edited record back to the same data file location it came from.

The data to be written is passed to <BTGetRec> via the <Source> parameter. Since <Source> is untyped, any desired data structure can be passed as a parameter.

When <ISOLock> is False, this procedure will operate successfully in a network environment only if the fileblock is locked for exclusive write access (a level 1 or 2 lock).

When <ISOLock> is True, the data record is written regardless of whether the fileblock is locked by either the calling workstation or any other workstation. Only a record lock (level 3) placed by another workstation can prevent writing the data record in this case.

Warning: Never add a new data record into the file with <BTPutRec>. Use <BTAddRec> to do that! Use <BTPutRec> only to update an existing record.

An attempt to update record 0 with <BTPutRec> will generate <IsamError> 10130. B-Tree Filer reserves record 0 for its own internal information.

Example

See Section 4.C.

See Also

BTAddRec

BTGetRec

BTReadLockAllOpenFileBlocks

Declaration

procedure BTReadLockAllOpenFileBlocks;

Purpose

Prevent all workstations from writing to open network fileblocks.

Description

This routine read locks all network fileblocks currently opened by the calling workstation. As a result, no other workstation can write to the fileblocks. (The only exception is that another station can still update individual data records by calling <BTPutRec> with <ISOLock> set to True.) All workstations can continue to read from the fileblocks as usual. Any number of workstations may simultaneously place read locks on the same fileblocks.

It is not possible to read lock any fileblock if another workstation has already obtained an exclusive write lock. <IsamError> will unlock all open fileblocks and return 10332 in this case.

<BTReadLockAllOpenFileBlocks> converts any write locks already obtained by the calling workstation into read locks, effectively removing the write locks. Remove read locks by calling <BTUnlockAllOpenFileBlocks> or <BTUnlockFileBlock>.

Example

```
BTReadLockAllOpenFileBlocks;  
if BTIsamErrorClass = 2 Then  
  WriteLn('Read lock attempt failed.');
```

A class 2 error code means that one of the read locks could not be completed because another workstation had already obtained a fileblock write lock.

See Also

BTLockAllOpenFileBlocks
BTUnlockAllOpenFileBlocks

BTReadLockFileBlock

BTReadLockFileBlock

Declaration

```
procedure BTReadLockFileBlock(IFBPtr : IsamFileBlockPtr);
```

Purpose

Prevent writing to the fileblock by another workstation.

Description

This routine read locks the specified fileblock, which means that no other workstation may write to it. (The only exception is that another station can still update individual data records by calling <BTPutRec> with <ISOLock> set to True.) All workstations can continue to read from the fileblock as usual. Any number of workstations may simultaneously place read locks on the same fileblock.

It is not possible to read lock a fileblock if another workstation has already obtained an exclusive write lock. <IsamError> will return 10332 in this case.

Redundant calls to <BTReadLockFileBlock> are not an error. The fileblock remains read locked.

<BTReadLockFileBlock> converts any write locks already obtained by the calling workstation into read locks, effectively removing the write locks. Remove a read lock by calling <BTUnlockFileBlock>.

B-Tree Filer uses temporary read locks internally to assure that index buffers remain stable while completing a sequence of operations. These internal read locks are placed and removed only if the fileblock is not already locked for reading or writing. An application can therefore achieve higher performance if it places its own read locks prior to a series of B-Tree Filer key operations. As with all locks, however, it is important to minimize the total time that the lock is maintained. Remember that no other workstation can write to the fileblock as long as the read lock remains active.

Example

```
BTReadLockFileBlock(PF);  
if BTIsamErrorClass = 2 then  
  Writeln('Read lock attempt failed');
```

A class 2 error code means that the read lock could not be completed because another workstation had already obtained a fileblock write lock.

See Also

BTLockFileBlock
BTUnlockFileBlock

BTReadLockAllOpenFileBlocks

BTRecIsLocked / BTSearchKey

Declaration

```
function BTRecIsLocked(IFBPtr : IsamFileBlockPtr; Ref : LongInt) : Boolean;
```

Purpose

Returns True if record <Ref> is locked.

Description

B-Tree Filer maintains an internal list of the records locked by each workstation. <BTRecIsLocked> returns True only if the current workstation locked the record. The only way to determine whether another workstation locked a record is to attempt to lock it from the current station and to check for success.

See Also

BTaRecIsLocked

BTLockRec

Declaration

```
procedure BTSearchKey(IFBPtr : IsamFileBlockPtr; Key : Word;  
                     var UserDatRef : LongInt; var UserKey : IsamKeyStr);
```

Purpose

Search for nearest matching key.

Description

The index file is searched for the key <UserKey> in the index <Key>. The matching key string is returned in <UserKey> and the data record reference is returned in <UserDatRef>. If an exact match is not found, the next larger key is returned in <UserKey> along with its associated record number in <UserDatRef>. The search fails and <IsamError> 10210 is returned only when the initial <UserKey> is larger than all other keys.

If a secondary key that is entered more than once in the index file is found, the smallest associated data record reference is returned in <UserDatRef>.

<BTSearchKey> sets the sequential pointer upon successful execution, so that the procedures <BTPrevKey> and <BTNextKey> may immediately be used.

<BTSearchKey> is implemented by first calling <BTFindKey>, and then <BTNextKey> if the first call was unsuccessful.

Example

See Sections 4.C and 4.F as well as the example for <BTNextKey>.

See Also

BTFindKey

BTNextKey

BTSearchKeyAndRef

BTSearchKeyAndRef

Declaration

```
procedure BTSearchKeyAndRef(IFBPtr : IsamFileBlockPtr; Key : Word;  
    var UserDatRef : LongInt;  
    var UserKey : IsamKeyStr);
```

Purpose

Search for a key near the given key and data record reference.

Description

Index <Key> of the index file is searched for the key <UserKey> that also has the record number <UserDatRef>. If it is found, the sequential pointer is set to point to this entry. If it is not found and there are any larger keys, the next larger key and record number are returned. If there are no larger keys, the next smaller key is returned. <IsamError> 10260 is returned if no match is found. This will occur only if the index is empty.

Example

```
var  
    Ref, SRef : LongInt;  
    IKS, SIKS : IsamKeyStr;  
    PTemp : PersonDef;  
    Stop : Boolean;  
...  
    BTFindKey(PF, 2, Ref, IKS);  
    Stop := False;  
    while IsamOK and not Stop do begin  
        BGetRec(PF, Ref, PTemp, False);  
        if IsamOK then  
            if PTemp.Age >= 18 then begin  
                SRef := Ref; SIKS := IKS; Stop := True;  
            end else  
                BNextKey(PF, 2, Ref, IKS);  
        end;  
        if BIsamErrorClass > 1 then begin  
            (Error handling)  
        end;  
        ...  
        BTSearchKeyAndRef(PF, 2, SRef, SIKS, 0);  
        if not IsamOK then begin  
            (Error handling)  
        end;  
    end;
```

The example first searches for a specific key of index 2. The while loop then scans for the first data record that has an Age field greater than or equal to 18 and saves the key string and data reference number for this record. Finally <BTSearchKeyAndRef> attempts to set the sequential pointer on the data record that was found in the while loop. It will succeed in returning a nearby key even if the original record was deleted in the meantime by another workstation.

See Also

BTSearchKey

BTFindKey

Declaration

```
function BTSetDosRetry(NrOfRetries, WaitTime : Integer) : Boolean;
```

Purpose

Set the number of retry attempts and timeout time for a read attempt aborted by a lock.

Description

This function makes sense only when the DOS file-sharing module SHARE.EXE is loaded or an equivalent network service is active. It requires DOS version 3.1 or later.

<BTSetDosRetry> calls DOS function \$440B to set the number of retries DOS should attempt when it detects that a file is locked during a read request. The delay between attempts is set to <WaitTime> "loop delays". A loop delay is equal to the time it takes for an assembly language LOOP instruction to run 65536 times. This delay varies from machine to machine.

<SetDosRetry> returns True unless the carry flag is set when the DOS function returns.

DOS will not restore the original values for <NrOfRetries> and <WaitTime> when a program ends. Generally, the application should restore the default system values (<NrOfRetries>=3 and <WaitTime>=1) before halting.

Experience shows that this function doesn't work as expected on every network. Some networks just ignore the retry values. However, the function is especially useful when using PC-MOS/386.

Example

PC-MOS/386 defaults for retries and delay seem to be quite high. As a result, when the B-Tree Filer BROWSER unit attempts to display a data record that another workstation has locked, the delay is annoyingly long. The following code is therefore recommended when initializing B-Tree Filer for PC-MOS/386:

```
if not BTSetDosRetry(1, 1) then begin
  {Error handling}
end;
```

BTSetSearchForSequential

Declaration

```
procedure BTSetSearchForSequential(IFBPtr : IsamFileBlockPtr;  
                                   Key : Word; On : Boolean);
```

Purpose

Sets the "SearchForSequential" option for index <Key> to <On>.

Description

This procedure activates or deactivates a special method for finding the next and previous keys from the current position in index <Key> of a fileblock. It primarily affects the behavior of the <BTNextKey> and <BTPrevKey> routines.

When you call <BTNextKey> (or <BTPrevKey>) the next key is computed based on the "current" position within the index. B-Tree Filer stores the current position by using a data structure referred to as the "sequential pointer". This is not a pointer in the usual sense of a physical memory address. Instead, the sequential pointer is an array of values that specify elements within a series of index pages, leading from the root page of the B-tree down to the page where the current key is located.

An alternate way to remember the current position would be to keep a copy of its key string and data reference number. B-Tree Filer doesn't use this approach. For one thing, in many cases the string would use more memory. More importantly, the sequential pointer approach is much faster because it retains exactly the information needed when time comes to move to the next or previous key; an additional search through the B-tree isn't required every time.

The disadvantage of the sequential pointer approach is that the pointer is specific to a particular configuration of the B-tree. If the B-tree is modified by adding or deleting a key, the numbers stored in the sequential pointer become meaningless. B-Tree Filer uses an internal flag to detect this condition. If you call <BTNextKey> or <BTPrevKey> in such a case, it will return with <IsamError> set to 10255 or 10265: sequential access not allowed.

For single user applications this behavior is acceptable, because the application knows exactly when it has disturbed the B-tree and can reinitialize the sequential pointer by calling <BTFindKeyAndRef> or a related key searching routine. In a network application, however, it is common for the sequential pointer to be corrupted by another workstation, which might call <BTAddKey> or <BTDeleteKey> at almost any time.

This is where the "SearchForSequential" option comes in. If the option is enabled and the flag indicating an invalid sequential pointer is set, <BTNextKey> will automatically use <UserKey> and <UserDatRef> (always passed to <BTNextKey> anyway) with <BTFindKey> to reinitialize the sequential pointer.

BTSetSearchForSequential

Whether the SearchForSequential option is enabled by default depends on the state of the global typed constant <SearchForSequentialDefault> interfaced by the FILER unit. <SearchForSequentialDefault> defaults to True, enabling the special search method for all fileblocks and indexes. Use <BTSetSearchForSequential> to disable the option for specific indexes.

See Also

BTAddKey

BTGetSearchForSequential

BTDeleteKey

BTNextKey

BTUnlockAllOpenFileBlocks

Declaration

procedure BTUnlockAllOpenFileBlocks;

Purpose

Unlock all open fileblocks.

Description

All open network fileblocks are unlocked. This reverses the action of <BTLockAllOpenFileBlocks> or <BTReadLockAllOpenFileBlocks>. At the same time all new data of all fileblocks locked for writing is flushed to disk.

Calling this procedure repeatedly is not an error. The network fileblocks remain unlocked. If an unlock fails for one of the fileblocks, an attempt is still made to unlock all others.

Example

```
BTUnlockAllOpenFileBlocks;  
if BTIsamErrorClass <> 0 then begin  
  WriteLn('Hard Error: Unlock attempt failed.');
```

(A more exact failure analysis could occur here.)

```
end;
```

- If <BTUnlockAllOpenFileBlocks> returns an error code, it will always be a hard error. Either data could not be written to disk, or a physical lock could not be removed.

See Also

BTLockAllOpenFileBlocks
BTUnlockFileBlock

BTReadLockAllOpenFileBlocks

BTUnlockAllRecs / BTUnlockFileBlock

Declaration

```
procedure BTUnlockAllRecs(IFBPtr : IsamFileBlockPtr);
```

Purpose

Unlock all locked records of the fileblock.

Description

B-Tree Filer maintains an internal list of the records locked by each workstation. <BTUnlockAllRecs> scans this list and releases each lock. If an error occurs while releasing one lock it continues to release the rest.

<BTUnlockAllRecs> removes only the locks placed by the current workstation, not by other workstations.

See Also

BTaRecIsLocked

BTUnlockRec

Declaration

```
procedure BTUnlockFileBlock(IFBPtr : IsamFileBlockPtr);
```

Purpose

Unlock the fileblock.

Description

The network fileblock is unlocked. This reverses the action of <BTLockFileBlock> or <BTReadLockFileBlock>. If the fileblock was locked for writing, all new data is flushed to disk.

Calling this procedure repeatedly is not an error. The fileblock remains unlocked.

Example

```
BTUnlockFileBlock(PF);  
if BTIsamErrorClass <> 0 then begin  
  Writeln('Hard Error: Unlock attempt failed.');
```

(A more exact failure analysis could occur here.)

```
end;
```

If <BTUnlockOpenFileBlock> returns an error code, it will always be a hard error. Either data could not be written to disk, or the physical lock could not be removed.

A more detailed example is found in Section 4.F.

See Also

BTReadLockFileBlock

BTLockFileBlock

BTUnlockRec

Declaration

```
procedure BTUnlockRec(IFBPtr : IsamFileBlockPtr; Ref : LongInt);
```

Purpose

Removes a lock from record <Ref>.

Description

A record lock (degree 3) is removed from the specified record. Calling this routine for a record that has already been unlocked is not an error. B-Tree Filer does not call the operating system to remove the lock again.

Example

See Section 4.F.

See Also

BTLockRec

BTUnlockFileBlock

BTUnlockAllRecs

Declaration

```
function BTUsedKeys(IFBPtr : IsamFileBlockPtr; Key : Word) : LongInt;
```

Purpose

Determine the number of keys added to index <Key>.

Description

B-Tree Filer keeps count of each key added or deleted from each index. Hence, this call doesn't need to scan the entire index; instead it gets the count by reading a single information record from the index file.

This function is especially useful with variable length records (when exactly one key is added for each logical record for a certain index), since there is no other fast method to determine the number of logical records.

Example

This function can be used to perform a simple consistency check on the data and index files if for every data record exactly one key was added.

```
var
  URecs : LongInt;
  UKeys : LongInt;
  I      : Word;
...
  URecs := BTUsedRecs(PF);
  if not IsamOK then begin
    {Error handling}
  end;
  for I := 1 To BTNrOfKeys(PF) do begin
    UKeys := BTUsedKeys(PF);
    if not IsamOK then begin
      {Error handling}
    end;
    if URecs <> UKeys then begin
      Writeln('Error! The number of data records is ', URecs);
      Writeln('The number of keys in index ', I, ' is only ', UKeys);
    end;
  end;
```

If an index doesn't have the same number of keys as there are data records, there must be some sort of error. This kind of error can be corrected with the REBUILD unit.

BTUsedRecs

Declaration

```
function BTUsedRecs(IFBPtr : IsamFileBlockPtr) : LongInt;
```

Purpose

Return the number of used data records.

Description

After a data record is deleted with the procedure <BTDeleteRec>, a "hole" remains in the data file. The "holes" represent space that will be filled by later calls to <BTAddRec>.

<BTUsedRecs> returns the number of actual data records (not counting deleted records) in the data file.

Example

```
var
  URecs : LongInt;
  DRecs : LongInt;
...
  URecs := BTUsedRecs(PF);
  if not IsamOK then begin
    {Error handling}
  end;
  DRecs := BTFreeRecs(PF);
  if not IsamOK then begin
    {Error handling}
  end;
  Writeln('The number of used records is ', URecs);
  Writeln('The number of deleted records is ', DRecs);
```

Displays the number of used and deleted records in the fileblock. Note that both routines must access information in the system record of the data file and therefore may generate errors.

See Also

BTFileLen

BTFreeRecs

6. B-Tree Utilities

These utilities make database applications even easier to write by building upon the FILER unit already described. Each utility is implemented as a Turbo Pascal unit that can be used whenever the program needs it. The utility units are designed to work in either single user or network mode--you receive all of these utilities no matter which version of B-Tree Filer you've purchased.

The following units are described in this chapter.

EMSHEAP

Memory can be allocated and freed like the conventional heap by using this expanded memory heap manager. The FILER unit uses EMSHEAP to manage index page buffers stored in EMS memory. You can use it to store your own data structures in expanded memory.

VREC

This unit provides special reading and writing routines for variable length data records. These routines are used in place of <BTAddRec>, <BTPutRec>, and <BTGetRec>. Among other uses, VREC enables the storage of "memo fields" (variable length notes) without wasting the space of the longest note for every record.

BROWSER

Display and scroll through fileblock records with this screen-oriented unit. The displayed format of each record is user-definable. The browser supports scroll bars and the mouse, is network-aware, and provides hooks for background tasks and user-defined commands. If you'd like a fileblock browser to work with Object Professional's windowing system, be sure to look at FBROWSELZH on the BONUS disk.

REBUILD/VREBUILD

These units regenerate the data and index files of a fileblock to remove deleted record space and create a clean index. Each key string is provided by a user-supplied routine. The units work with fixed and variable length records.

REORG/VREORG

These units are similar to REBUILD and VREBUILD, but provide the additional capability of modifying each data record as it is passed from input to output. Use them to import data from foreign fixed-length record formats or to add/delete fields from existing B-Tree Filer fileblocks. Data record modification occurs within a user-supplied routine.

FIXTOVAR

Convert a fixed length record fileblock to a variable length record fileblock with this unit.

NUMKEYS

This unit provides a variety of functions for manipulating key strings, including routines to convert numbers to and from strings, and routines to compress ASCII strings, yielding up to 50% reduction in index size.

ISAMTOOL

This is a collection of miscellaneous routines to extend the **FILER** unit, which for various reasons were felt to be unsuitable for inclusion in the unit itself. Included are routines for increasing the number of file handles available to a program, assigning a unique workstation identification number in a network-independent manner, returning a text string for each <IsamError> code, and inverting a key string to provide for descending indexes.

MSORT

Any type and number of data items can be sorted with this unit, which implements a merge sort algorithm that can utilize disk space or EMS for secondary storage. Control of input, comparison criteria, and output is provided by user-supplied routines.

A. EMSHEAP: Using EMS for Heap Storage

EMSHEAP offers the same basic services as Turbo Pascal's built-in heap manager, with the exception that EMSHEAP uses EMS memory for its source of storage space. B-Tree Filer's FILER unit calls upon EMSHEAP to manage the storage of index page buffers in EMS memory; for that application you won't need to know the details of EMSHEAP's operation. You can also call EMSHEAP directly if your data storage requirements exceed the capacity of the DOS 640K memory limit.

EMSHEAP offers a small but powerful set of routines for managing EMS:

<GetEMSMem> and **<FreeEMSMem>**

allocate and deallocate blocks of EMS, with a minimum granularity of 64 bytes and a maximum block size of 32K bytes

<MapEMSPtr>

converts a logical pointer returned by **<GetEMSMem>** into a physical pointer usable in normal Pascal expressions

<EMSMemAvail> and **<EMSMaxAvail>**

return available EMS memory like their Turbo Pascal counterparts

<SaveEMSCtxt> and **<RestoreEMSCtxt>**

save and restore the state of the EMS page frame to avoid disturbing other users of EMS memory

<InitEMSHeap> and **<ExitEMSHeap>**

initialize and dispose of EMS memory management structures at runtime (optional)

We'll describe how to configure EMSHEAP, guidelines for using it, and some of its limitations in the sections that follow.

EMSHEAP depends on the services of an expanded memory driver compatible with Intel specification version 3.2 or later. The source of physical memory for the EMS is not important, except to the extent that it determines performance. For example, EMS that is implemented with mapped extended memory on an 80386 machine (using, for example, a 386MAX or QEMM386 driver) is usually the fastest. EMS implemented with a separate hardware card (such as an Intel AboveBoard) is as fast as the 386 or perhaps a little slower. Significantly slower is an EMS driver that emulates expanded memory using 80286 extended memory, since memory blocks must be frequently copied to and from extended memory. And of course the slowest EMS driver is one that uses disk space to emulate expanded memory.

EMSHEAP isolates itself from direct calls to the expanded memory manager by using routines in the EMSSUPP unit, which is also supplied. EMSSUPP is patterned

after the TPEMS and OPEMS units from Turbo Professional and Object Professional, respectively, so that you can replace EMSSUPP with these units if you're already using them in your program. Since EMSSUPP is not designed for direct use, we won't document it here.

You may wish to skip the following few paragraphs if you're already familiar with EMS memory.

The EMS standard developed by Lotus, Intel, and Microsoft is defined by the following basic characteristics:

- a separately addressed RAM area of up to 32 megabytes, addressable in 16K byte logical pages
- a "page frame" that typically consists of 64K bytes of addressing space located in the top portion of normal memory. The 32 megabytes of EMS memory are accessed by mapping one 16K block at a time into each of 4 "physical pages" (numbered 0..3) that comprise the page frame
- a device driver that isolates the application from the EMS hardware. This program, often called the EMM (expanded memory manager), will be referred to here as the EMS driver

The basic functions of the EMS driver are to manage allocation and deallocation of the logical pages, and to map the logical pages into the physical page frame for access by the application.

A group of logical pages is assigned a handle number for identification when it is allocated by the EMS driver. The EMS driver uses the handle number when it maps this group of logical pages into the page frame. At the same time, the EMS driver requires the logical page number within the group (starting from 0 for each group) and the physical page number (0..3) to map it into.

This is not the only use of an EMS handle. The EMS driver can remember a particular context for the page frame (which logical pages and handles are currently mapped into it) and restore it later. Each such context must be associated with a given EMS handle. As a result, the number of available context mappings is equal to the number of EMS handles allocated, which has direct consequences on the configuration of EMSHEAP, as you'll see below.

Configuration of EMSHEAP

Before adding EMSHEAP to the uses list of a program or another unit, you should check and perhaps modify BTDEFINE.INC to specify the desired compilation defines. The following defines have an effect on EMSHEAP. Note that you activate

a define by removing the period between the '{' and '\$', and deactivate it by inserting a period there.

UseTPEMS

UseOPEMS

By defining one or the other of these conditionals, you can save about 5.5K bytes of code if your application already uses the TPEMS or OPEMS units of Turbo Professional or Object Professional. This prevents the use of the EMSSUPP unit, and TPEMS or OPEMS is used instead.

DebugEMSHeap

NoErrorCheckEMSHeap

DebugEMSHeap should be used only if problems occur with the EMSHEAP unit. It adds checks within the EMSHEAP routines which increase the size of the unit. When an error occurs, these checks display an error code and in some cases abort the program. DebugEMSHeap activates error checking within the initialization block of EMSHEAP: if no EMS driver is found, the EMS version is too old, not enough EMS memory is available, or some other initialization error occurs, an error code will be reported but the application will continue (without access to EMS, of course). It modifies the error checking of <GetEMSMem>; the application will halt if not enough memory is available. It also activates a check for mapping the Nil pointer in <MapEMSPtr>.

Definition of the symbol NoErrorCheckEMSHeap disables certain standard error checks, which makes the EMSHEAP unit as fast as possible. This should be done only after your application is known to be bug free.

ManualInitEMSHeap

Define ManualInitEMSHeap if you wish to specify the configuration of EMS memory at runtime. ManualInitEMSHeap is not defined by default; in this case the EMSHEAP unit configures itself automatically. In some cases, especially when your program uses other units that consume EMS space, it may be desirable to decide how much EMS to allocate at runtime. See procedure <InitEMSHeap> for more information.

The file EMSHEAP.CFG also contains four constants that control the configuration of the EMSHEAP unit.

```
HandlesToUseForAlloc = 8;
```

As mentioned above, the number of EMS handles allocated determines how many page frame mapping contexts can be stored simultaneously. Therefore the default value of <HandlesToUseForAlloc> will suffice for up to eight unrelated mappings. If EMS is used extensively, a larger number of available mappings will lead to better performance. The acceptable range for <HandlesToUseForAlloc> is from 1

to 252. The lower value of 1 must be raised to 2 if EMSDisturbance is defined in BTDEFINE.INC (because the FILER unit then reserves one handle to preserve its own page mapping context). The upper limit of 252 is set by the EMS driver's limit of 255 after accounting for handles that EMSHEAP uses internally.

In addition, the constant <HandlesToUseForAlloc> influences the minimum and maximum possible size of the EMS heap. Note the description for the following two constants.

```
MinEMSHeapPages = HandlesToUseForAlloc;  
MaxEMSHeapPages = 2048;
```

These constants determine the minimum and maximum number of logical pages that may be allocated for the EMS heap. Each page corresponds to 16K bytes of EMS memory. <MinEMSHeapPages> must be at least as large as <HandlesToUseForAlloc>, since each handle must allocate at least one page. <MaxEMSHeapPages> must be at least as large as <MinEMSHeapPages>. The maximum value for both constants is 2048, which corresponds to the 32 megabyte maximum of EMS memory.

Normally these constants don't need to be modified. However, if a program absolutely requires a certain minimum amount of EMS memory, <MinEMSHeapPages> could be set to this value. If the computer has less than <MinEMSHeapPages> of free EMS available, the EMSHEAP initialization block sets the global boolean variable <EMSHeapInitialized> to False. If a program wants to reserve some EMS space for uses besides EMSHEAP, it can set <MaxEMSHeapPages> to limit the space EMSHEAP will use. However, it's better to use <ToLetFreePages> for this purpose.

```
ToLetFreePages = 0;
```

This constant determines the number of EMS pages to leave free after EMSHEAP initialization is complete. If left at the default value of 0, EMSHEAP will allocate all available EMS memory, up to <MaxEMSHeapPages>. The maximum acceptable value for <ToLetFreePages> is

$$(\text{Number of available logical pages}) - 4 - \text{<MinEMSHeapPages>}$$

That is, EMSHEAP must be able to allocate at least <MinEMSHeapPages> pages plus four more pages that it uses to maintain an internal free list of disposed blocks.

When ManualInitEMSHeap is undefined, the EMSHEAP initialization code allocates as many logical pages as it can, consistent with these constants. EMSHEAP then manages suballocation of requested blocks from these pages. For internal reasons, EMSHEAP cannot give more than 4 megabytes to one EMS handle, so the maximum number of bytes in the EMS heap will be

<HandlesToUseForAlloc> * 4194304

As mentioned above, EMSHEAP allocates at least one logical page for each EMS handle, so the minimum number of bytes in the EMS heap will be

<HandlesToUseForAlloc> * 16384

Hence, using the default constants, at least 192K bytes of EMS (8*16K + 64K for the free list) must be free for successful initialization of EMSHEAP. If initialization fails, <EMSHeapInitialized> will be set to False.

In all cases EMSHEAP uses no memory from the normal heap. It does consume a small amount of space in the data segment, where memory use can be calculated as follows:

$60 + 13 * \text{<HandlesToUseForAlloc>}$

For the default settings this amounts to 164 bytes of data segment space.

The granularity of the EMS heap is set to 64 bytes, primarily to minimize fragmentation. All block sizes passed to <GetEMSMem> and <FreeEMSMem> are automatically adjusted to a multiple of 64.

The maximum size of any one memory block allocated by a program that uses EMSHEAP is 32K bytes. The most important reason for this limitation is that it allows copying directly from one EMS memory block to another without intermediate page mapping or use of an intermediate buffer.

Note that EMSHEAP installs a Turbo Pascal exit procedure: when the program halts (normally or with a runtime error), EMSHEAP automatically frees all EMS memory that it allocated. There is one important caveat: if you are running your program under Turbo Debugger and stop the program by exiting the debugger, the program's exit procedures are not executed, and EMS memory will remain allocated. In this case, you'll need to reboot your computer to regain access to this memory.

Coexisting with Turbo Pascal's OVERLAY Unit

The OVERLAY unit provided with Turbo Pascal can benefit greatly from the availability of EMS. The overlay manager allocates EMS at the time of the call to OvrInitEMS. In order to assure friendly coexistence between EMSHEAP and OVERLAY, there are three cases to consider:

1. If ManualInitEMSHeap is undefined, and OvrInitEMS is to be called at any time after EMSHEAP's initialization block is executed, you must prevent EMSHEAP from allocating all available EMS space. You can do so by setting <ToLetFreePages> to an appropriate value, typically equal to the size

of the OVR file generated by the compiler, rounded up to the next even multiple of 16K, divided by 16K. The drawback to this method is that <ToLetFreePages> must be adjusted for each overlay file and each time the overlay file changes size. As a result, we recommend one of the following approaches instead.

2. If ManualInitEMSHeap is defined, then <InitEMSHeap> should be called with a parameter of <FreePages> large enough to leave space for the overlay. Then OvrInitEMS may be called.
3. If OvrInitEMS is called prior to the initialization block of EMSHEAP, then OVERLAY can allocate the space that it needs in EMS, leaving the remainder free for the EMSHEAP unit. In order to do this, OvrInitEMS must be called from the initialization block of another unit, and that unit must be listed in the uses statement of the main program before EMSHEAP or any unit that uses EMSHEAP. The Turbo Pascal manual describes the initialization of the overlay manager in this fashion; see the Borland manual for more information.

User Hooks for Hard Errors and Allocation Failures

EMSHEAP's default action upon encountering a hard error--such as calling an EMSHEAP routine when <EMSHeapInitialized> is False or failure of a call made to the EMS driver--is to display the error number on the screen and to abort the program. The following typed constant allows modification of the default action:

```
EMSHardErrorFuncPtr : Pointer = Nil;
```

In order to provide a non-default action, assign <EMSHardErrorFuncPtr> the address of a function that you have written. This function must be compiled with the far model, must be global, and must match the following prototype declaration:

```
function MyHardErrorFunc(Error : Word) : Boolean;
```

When the error function is called, <Error> contains the code for the error that just occurred (error codes are described below). The function should return True to abort the program, or False to continue. Program continuation after an EMS hard error is not generally recommended; actions should usually be limited to a graceful shutdown of non-EMS program operations.

A hard error function can be installed with the following statement:

```
EMSHardErrorFuncPtr := @MyHardErrorFunc;
```

In order to return to the default action, just use:

```
EMSHardErrorFuncPtr := Nil;
```

Warning: the hard error function may not call any routine of the EMSHEAP unit either directly or indirectly.

The default action of <GetEMSMem> when called with a request for more bytes of EMS memory than are available is to return a Nil pointer. The following typed constant allows modification of the default action:

```
EMSHeapErrorFuncPtr : Pointer = Nil;
```

<EMSHeapErrorFuncPtr> is analogous to Turbo Pascal's HeapError function pointer. Whenever a <GetEMSMem> request cannot be fulfilled, EMSHEAP calls the corresponding function. <EMSHeapErrorFuncPtr> contains the address of a function that must be compiled with the far model, must be global, and must match the following prototype declaration:

```
function MyHeapErrorFunc(Size : Word) : Integer;
```

When the EMS heap error function is called, <Size> contains the number of bytes (after rounding to a 64 byte boundary) that were requested but could not be allocated. The function returns one of three values:

- 0 Call the hard error function
- 1 Return a Nil pointer
- 2 Retry the operation

When <EMSHeapErrorFuncPtr> is Nil, EMSHEAP's behavior is equivalent to the error function returning a 1. Note the contrast to Turbo Pascal's normal heap error function, which by default returns 0 to halt the application.

Install a non-default EMS heap error function with a statement like the following:

```
EMSHeapErrorFuncPtr := @MyHeapErrorFunc;
```

Warning: The heap error function may call only the following routines from the EMSHEAP unit: <GetEMSMem>, <FreeEMSMem>, <EMSMemAvail>, and <EMSMaxAvail>.

Issues When Using EMSHEAP

For the most part, EMSHEAP works like the normal Turbo Pascal heap and the calls you make to it are handled the same way. However, there are three important differences caused by the nature of the compiler and the nature of EMS memory.

The first difference is that there is no EMSHEAP counterpart for New and Dispose. These procedures are more like compiler macros, since they push a hidden parameter that specifies the size to allocate before calling an internal system allocation routine. When used for objects, New and Dispose are even trickier, since these operations actually occur within the constructor or destructor of the object. EMSHEAP routines can be used to replace New and Dispose for non-object applications simply by passing the size of the block to be allocated explicitly to the routine.

The second difference occurs because all EMS memory is accessed through the tiny 64K page frame. Hence, only a small part of EMS is immediately available and thereby useful. Before the data can be used, it must be mapped into the page window. The mapping operation occurs at the time when <MapEMSPtr> converts a "logical" pointer returned by <GetEMSMem> to a "physical" pointer usable by other Turbo Pascal routines. Since every call to <MapEMSPtr> may remap the page frame, there's no guarantee that the pointer returned by a prior call to <MapEMSPtr> remains valid. As a result, the application program must take care to assure that physical pointers to the EMS page frame remain valid at the time when they are used.

The simplest and safest method to assure validity is to call <MapEMSPtr> immediately before using each EMS pointer. Be sure to realize the consequences of choosing this strategy: No pointer returned by <MapEMSPtr> may be passed as a parameter to a routine that also calls <MapEMSPtr>. No pointer returned by <MapEMSPtr> may be used as the argument to a With statement within which <MapEMSPtr> is called. No pointer returned by <MapEMSPtr> may be assigned to a variable that is used after another call to <MapEMSPtr>.

This simple method is not usually optimal because it generates unneeded calls to <MapEMSPtr>, which reduce performance. (Note, however, that <MapEMSPtr> does detect when the page frame is already mapped correctly and does not call the EMS driver unnecessarily.) An optimal method is available, but it requires a more detailed understanding of how EMSHEAP works. The FILER unit optimizes its calls to <MapEMSPtr> in this way.

The basis for the optimization is knowing the life span of a page mapped via <MapEMSPtr>. The life span is dependent upon the size of the mapped block and the mappings that follow it. Since the largest possible block allocated via <GetEMSMem> is 32K bytes, blocks may consist of at most two EMS pages. EMSHEAP uses the following rules to decide how to remap the page frame for a given series of calls to <MapEMSPtr>:

1. If memory blocks all smaller than or equal to 16K bytes are mapped, the last four mappings remain valid.
2. If memory blocks whose size is greater than 16K bytes (and less than or equal to 32K bytes, the maximum) are mapped one immediately after the other, the last two mappings remain valid.
3. If a block larger than 16K bytes is mapped just after a block smaller than 16K bytes, all previous mappings become invalid.
4. If a block smaller than 16K bytes is mapped just after a block larger than 16K bytes, the larger block's mapping remains valid. Another block smaller than 16K may also be mapped, leaving three mappings valid.

The following example program may help to make these rules more tangible.

```

uses
  EMSHeap;
type
  MyArraySmall = array[1..16000] of Byte;
  SmallPtr      = ^MyArraySmall;
  MyArrayBig    = array[1..32000] of Byte;
  BigPtr        = ^MyArrayBig;
var
  I      : Integer;
  EMSPtr1 : array[1..5] of EMSPtr;
  EMSPtr2 : array[1..3] of EMSPtr;
  DataPtr1 : array[1..5] of SmallPtr;
  DataPtr2 : array[1..3] of BigPtr;
begin
  for I := 1 To 5 do begin
    GetEMSMem(EMSPtr1[I], SizeOf(MyArraySmall));
    if EMSPtr1[I] = Nil then begin
      Writeln('Not enough memory available.');

```

Five sections of 16000 bytes are allocated on the EMS heap. The logical pointers of type <EMSPtr> returned by <GetEMSMem> are stored in the pointer array EMSPtr1. <MapEMSPtr> is then used to map the data area into the EMS frame and return a physical pointer to this area. It must be typecast to the proper type (SmallPtr) since <MapEMSPtr> returns a generic pointer. The area pointed to by DataPtr[I]^ is then filled with a value.

How should the EMS pointers be used later in the program? First, note that the last four mapped pointers, DataPtr[2] through DataPtr[5], are still valid after the For loop completes. Therefore a statement like

```
DataPtr1[3]^ := DataPtr1[2]^;
```

is valid without another call to <MapEMSPtr>. By contrast, the following statement will copy the wrong data into DataPtr1[3]^:

```
DataPtr1[3]^ := DataPtr1[1]^;
```

Now assume that the main block of the example is replaced with the following code which allocates blocks larger than 32K:

```
for I := 1 To 3 do begin
  GetEMSMem(EMSPtr2[I], SizeOf(MyArrayBig));
  if EMSPtr2[I] = Nil then begin
    Writeln('Not enough memory available');
    Halt;
  end;
  DataPtr2[I] := BigPtr(MapEMSPtr(EMSPtr2[I]));
  FillChar(DataPtr2[I]^, SizeOf(MyArrayBig), 1);
end;
{Further use of the initialized sections}
```

Now only DataPtr2[2] and DataPtr2[3] are valid for the further use. DataPtr2[1] points to an undefined area.

Finally, assume that both the EMSPtr1 and EMSPtr2 arrays have been allocated, leading to a mix of blocks with sizes less than and greater than 16K, and consider the following sequence of calls to <MapEMSPtr>:

```
DataPtr1[1] := SmallPtr(MapEMSPtr(EMSPtr1[1]));
DataPtr1[2] := SmallPtr(MapEMSPtr(EMSPtr1[2]));
DataPtr1[3] := SmallPtr(MapEMSPtr(EMSPtr1[3]));
DataPtr1[4] := SmallPtr(MapEMSPtr(EMSPtr1[4]));
{Checkpoint 1}
DataPtr2[1] := BigPtr(MapEMSPtr(EMSPtr2[1]));
{Checkpoint 2}
DataPtr2[2] := BigPtr(MapEMSPtr(EMSPtr2[2]));
{Checkpoint 3}
DataPtr1[1] := SmallPtr(MapEMSPtr(EMSPtr1[1]));
DataPtr1[2] := SmallPtr(MapEMSPtr(EMSPtr1[2]));
{Checkpoint 4}
```

At checkpoint 1, all four values in the DataPtr1 array remain valid (this is an example of rule 1). At checkpoint 2, only DataPtr2[1] is valid (rule 3). At checkpoint 3, both DataPtr2[1] and DataPtr2[2] are valid (rule 2). At checkpoint 4, DataPtr2[2], DataPtr1[1], and DataPtr1[2] are valid (rule 4).

Fairly complicated rules such as these may lead you to choose the simpler method of just calling <MapEMSPtr> every time. Your choice will depend on the importance of EMS access speed for your application. Remember that EMSHEAP is unable to detect failure on your part to follow the mapping rules; the application will just use the wrong data leading to unpredictable consequences.

The third difference is also caused by the nature of page frame mapping. This issue arises only when two or more different EMS users are accessing the EMS page frame within the same system. In this case, it's important that the two users don't disturb one another, i.e., that one or both of the users leave the page frame undisturbed.

A common example occurs when a disk cache program that uses EMS is loaded while a program that also uses EMS is accessing the disk. The disk cache is responsible for saving the page frame context before changing it, and then restoring the page frame before returning control to the application. This is essential because the application would have no way of knowing when its page frame was disturbed. Another common example would occur when the Turbo Pascal OVERLAY unit is enabled for EMS usage and the EMSHEAP unit is also used. The OVERLAY unit is designed to save and restore the EMS context, so it won't disturb EMSHEAP.

Even though EMSHEAP need not save and restore context for these two examples, it's still a good idea for it to do so in general, since other cases may arise where it's essential. (Such a case would occur if EMSHEAP and the EMS array routines from Turbo Professional or Object Professional.) To serve this need, EMSHEAP offers the <SaveEMSCtxt> and <RestoreEMSCtxt> procedures. If the example program shown above were to be localized in a function, it would use the context saving routines as follows:

```

function DoSomethingWithData : Boolean;
type
  MyArraySmall = array[1..16000] of Byte;
  SmallPtr     = ^MyArraySmall;
  MyArrayBig   = array[1..32000] of Byte;
  BigPtr       = ^MyArrayBig;
var
  I           : Integer;
  EMSPtr1     : array[1..5] of EMSPtr;
  EMSPtr2     : array[1..3] of EMSPtr;
  DataPtr1    : array[1..5] of SmallPtr;
  DataPtr2    : array[1..3] of BigPtr;
  SaveHandle  : Byte;
begin
  DoSomethingWithData := False;
  SaveHandle := SaveEMSCtxt;
  for I := 1 To 5 do begin
    GetEMSMem(EMSPtr1[I], SizeOf(MyArraySmall));
    if EMSPtr1[I] = Nil then begin
      RestoreEMSCtxt(SaveHandle);
      Exit;
    end;
    DataPtr1[I] := SmallPtr(MapEMSPtr(EMSPtr1[I]));
    FillChar(DataPtr1[I]^, SizeOf(MyArraySmall), 1);
  end;
  {Further use of the initialized sections}
  RestoreEMSCtxt(SaveHandle);
  DoSomethingWithData := True;
end.

```

The function returns False if the required EMS memory is not available, but even in this case it restores the page frame before returning. (The example does not deallocate the EMS memory already allocated, which it should do in a real application.) Before the routine exits, it calls <RestoreEMSCtxt> to restore the page frame.

Note that calls to <GetEMSMem> or <FreeEMSMem> do not disturb the page frame. If this example routine never called <MapEMSPtr>, there would be no need for it to call <SaveEMSCtxt> or <RestoreEMSCtxt>.

The context management routines are described more thoroughly in the reference section that follows.

Error Codes

Some of the following error codes are generated only if particular EMSHEAP conditional defines are activated. Such codes are indicated by {\$IFDEF Symbol} or {\$IFNDEF Symbol} in the table.

If DebugEMSHeap is defined, initialization errors are reported when they occur with a routine that calls Writeln, then waits for <Enter> to be pressed. Execution of the program continues, but <EMSHeapInitialized> is set to False.

If DebugEMSHeap is not defined, no error code is reported, but <EMSHeapInitialized> is set to False.

When an error occurs after initialization, EMSHEAP calls the hard error routine, unless the error was the result of insufficient free EMS space, in which case it calls the allocation error routine. See the section entitled "User Hooks for Hard Errors and Allocation Failures" above for more information.

Error codes during initialization (all {\$IFDEF DebugEMSHeap})

- 1 EMS driver not found
- 2 EMS version is not 3.2 or greater
- 3 Pointer to the EMS page frame could not be determined
- 4 Number of usable EMS pages could not be determined
- 5 <HandlesToUseForAlloc> is less than <MinEMSHeapPages>
- 6 The number of free EMS pages is less than <MinEMSHeapPages>
- 7 Too many pages were requested for one EMS handle. The maximum is 256 pages (4 megabytes per EMS handle). <HandlesToUseForAlloc> needs to be increased.
- 8 EMS memory could not be allocated
- 9 EMS page frame context could not be saved
- 10 Mapping a logical EMS page was not possible
- 11 Not enough EMS handles to fulfill memory request
- 12 EMS page frame context could not be saved
- 13 EMS page frame context could not be restored
- 20 Invalid page frame pointer (offset is not 0)

Other errors

- 50 {\$IFDEF NoErrorCheckEMSHeap} An EMSHEAP routine was called while <EMSHeapInitialized> was False
- 60 {\$IFDEF NoErrorCheckEMSHeap} A disallowed EMSHEAP routine was called from within the EMSHeapError function
- 70 {\$IFDEF ManualInitEMSHeap} Manual initialization is possible only if the ManualInitEMSHeap directive is defined
- 75 {\$IFDEF ManualInitEMSHeap} Manual initialization has already been performed
- 80 {\$IFDEF ManualInitEMSHeap} Manual deinitialization is possible only if the ManualInitEMSHeap directive is defined

- 100 EMS page frame context could not be saved before mapping in the free list
- 101 Mapping the free list was not possible
- 102 EMS page frame context could not be saved before unmapping the free list
- 103 EMS page frame context could not be restored after unmapping the free list
- 110 EMS page frame context could not be saved in <SaveEmsCtxt>
- 111 <SaveEmsCtxt> failed because no free EMS handles were available.
Increase <HandlesToUseForAlloc>.
- 112 EMS page frame context could not be restored in <RestoreEmsCtxt>
- 120 {\$IFDEF NoErrorCheckEMSHeap} Attempt to allocate a block larger than 32K bytes
- 121 Insufficient free EMS is available
- 130 {\$IFDEF NoErrorCheckEMSHeap} The pointer passed to <FreeEMSMem> is Nil
- 131 {\$IFDEF NoErrorCheckEMSHeap} The size passed to <FreeEMSMem> is greater
than 16K bytes, and the corresponding pointer doesn't match
- 132 {\$IFDEF NoErrorCheckEMSHeap} The size passed to <FreeEMSMem> is less than
16K bytes, and the corresponding pointer doesn't match
- 140..145 Mapping a logical EMS page was not possible
- 150 {\$IFDEF DebugEMSHeap} The virtual pointer passed to <MapEMSPtr> is Nil
- 200 Mapping a logical page via <MapEMSPtr> is not possible (internal error)
- 201 Mapping a logical page via <MapEMSPtr> is not possible (internal error)

Constants

For further information about these constants, see the sections entitled "User Hooks for Hard Errors and Allocation Failures" and "Configuration of EMSHEAP" above.

```
DoManualInitEMSHeap =  
  {IFDEF ManualInitEMSHeap}  
    True;  
  {ELSE}  
    False;  
  {ENDIF}
```

This constant reflects the state of the ManualInitEMSHeap define. If <DoManualInitEMSHeap> is False, you should not call <InitEMSHeap> or <ExitEMSHeap>.

```
EMSHardErrorFuncPtr : Pointer = Nil;
```

Address of the function for hard errors. The default value of Nil disables a user defined function.

```
EMSHeapErrorFuncPtr : Pointer = Nil;
```

Address of the heap error function which is called when memory requests cannot be fulfilled. The default value of Nil disables a user defined function.

```
HandlesToUseForAlloc = 8;
```

Number of EMS handles used to allocate the EMS heap. Valid values are between 1 and 252.

```
MinEMSHeapPages = HandlesToUseForAlloc;
```

Minimum number of logical pages that must be available to initialize EMSHEAP successfully. Valid values are between <HandlesToUseForAlloc> and 2048.

```
MaxEMSHeapPages = 2048;
```

Maximum number of logical pages that EMSHEAP may use. Valid values are between <MinEMSHeapPages> and 2048.

```
ToLetFreePages = 0;
```

Number of logical pages that EMSHEAP will leave free at initialization, for the use of another application or another part of the program. <ToLetFreePages> must be small enough so that at least <MinEMSHeapPages>+4 pages can be allocated by EMSHEAP.

Types

EMSPtr = Pointer;

All logical pointers returned by <GetEMSMem> or passed to <FreeEMSMem> are of this type. This pointer may not be used directly, but must always be passed to <MapEMSPtr> which will return a physical pointer.

Variables

EMSHeapInitialized : Boolean;

This variable is set by the initialization code of the EMSHEAP unit and should not be modified by the user program. If <EMSHeapInitialized> is False, no routines from EMSHEAP may be called. To determine the reason for a False result, enable the DebugEMSHeap define.

Declaration

function EMSMaxAvail : Word;

Purpose

Returns the size of the largest available block.

Description

The return value will always be between 0 and 32K bytes, since EMSHEAP does not allow blocks larger than 32K. A following call to <GetEMSMem> will be successful if called with a size less than or equal to the value returned by <EMSMaxAvail>.

Example

```
var
  MyArray      : array[1..20000] of Byte;
  MyArEMSPtr   : EMSPtr;
...
  if EMSMaxAvail >= SizeOf(MyArray) then
    GetEMSMem(MyArEMSPtr, SizeOf(MyArray))
  else
    Writeln('Not enough EMS memory.');
```

The call to <EMSMaxAvail> determines that there is still enough free memory to allocate MyArray in the EMS heap. By checking <EMSMaxAvail> first, there is no concern that <GetEMSMem> will activate the EMS heap error function and no need to check whether <GetEMSMem> returns a Nil pointer. On the other hand, by calling <EMSMaxAvail> first we end up scanning the EMS free list twice, so this code may be slower than calling <GetEMSMem> and testing whether it failed.

See Also

EMSMemAvail

GetEMSMem

EMSMemAvail

Declaration

```
function EMSMemAvail : LongInt;
```

Purpose

Returns the total amount of free EMS space.

Description

If you call <EMSMemAvail> before calling <GetEMSMem> the first time, you'll get the total size of the EMS heap. The total size will always be between <MinEMSHeapPages>*16KB and <MaxEMSHeapPages>*16KB.

Note that <GetEMSMem> might fail even if <EMSMemAvail> returns a value larger than the needed block size. That's because the total returned by <EMSMemAvail> may be composed of many small non-contiguous pieces of EMS.

Example

```
Writeln('Free memory in the EMS heap: ', EMSMemAvail, ' Bytes');  
Displays available space on the EMS heap.
```

See Also

EMSMaxAvail

Declaration

```
procedure ExitEMSHeap;
```

Purpose

Deinstall the EMSHEAP unit.

Description

This routine frees all EMS pages allotted to EMSHEAP. Nothing happens if EMSHEAP has already been deinstalled. The boolean variable <EMSHeapInitialized> is set to False after successful completion.

Calling this routine when <DoManualInitEMSHeap> has a value of False generates a call to the hard error function with an error code of 80.

If EMSHEAP is being used in combination with the FILER unit, do not call <ExitEMSHeap> until after you have called <BTExitIsam>.

<ExitEMSHeap> is called automatically by the EMSHEAP unit's exit procedure.

Example

```
if DoManualInitEMSHeap then  
  ExitEMSHeap;
```

Deinstalls the EMSHEAP unit at the end of a program.

See Also

InitEMSHeap

FreeEMSMem

Declaration

```
procedure FreeEMSMem(EPtr : EMSPtr; Size : Word);
```

Purpose

Frees a previously allocated section of the EMS heap.

Description

<EPtr> is a logical pointer that was obtained through a previous call to <GetEMSMem>. <Size> must be exactly the same as originally allocated.

<FreeEMSMem> updates the internal list of free blocks. If the new free block adjoins free blocks on one or both sides of itself, those free blocks are merged into one. Hence, if all allocated EMS space is freed by calling <FreeEMSMem>, the EMS heap will return to the same state it had after initialization.

Example

```
var
  MyArray   : array[1..20000] of Byte;
  MyArEMSPtr : EMSPtr;

  ...
  GetEMSMem(MyArEMSPtr, SizeOf(MyArray));
  {Error checking}
  ...
  FreeEMSMem(MyArEMSPtr, SizeOf(MyArray));
```

Note that the variable MyArEMSPtr is invalid and may not be passed to <MapEMSPtr> after the call to <FreeEMSMem>.

See Also

GetEMSMem

Declaration

```
procedure GetEMSMem(var EPtr : EMSPtr; Size : Word);
```

Purpose

Allocates a memory block on the EMS heap.

Description

<Size> is the size of the block to allocate, which must be in the range of 1 to 32768, inclusive. If <Size> is zero, <GetEMSMem> sets <EPtr> to Nil and exits without generating an error. If <Size> is larger than 32768, <GetEMSMem> generates hard error 120 and exits. All <Size> values are then adjusted to a multiple of 64 bytes.

<EPtr> returns a "logical" pointer to the allocated block. This logical pointer may only be passed to <MapEMSPtr> or <FreeEMSPtr>; it should never be used in a Pascal expression directly.

<GetEMSMem> scans the free list of EMS blocks to find a block large enough to hold the requested block. It takes the first sufficiently large block that it finds. (Requested blocks larger than 16K may actually reside within two non-adjacent logical EMS pages.) If <GetEMSMem> cannot find sufficient space, its behavior depends on a heap error function. By default, <GetEMSMem> returns <EPtr> set to Nil in this case. See "User Hooks for Hard Errors and Allocation Failures" above for more information.

If you call <EMSMaxAvail> to determine whether enough memory is available first, you can guarantee (short of a sudden hardware breakdown) that <GetEMSMem> will succeed.

Example

```
GetEMSMem(MyArEMSPtr, SizeOf(MyArray));  
if MyArEMSPtr = Nil then  
  Writeln('Not enough EMS memory.');
```

Contrast this example to the one shown for <EMSMaxAvail>. This approach is generally faster but assumes that the default heap error function, or another one with similar behavior, is used.

See Also

EMSMaxAvail

FreeEMSMem

InitEMSHeap

Declaration

```
procedure InitEMSHeap(FreePages : Word);
```

Purpose

Explicitly initialize the EMSHEAP unit.

Description

<InitEMSHeap> initializes the EMSHEAP unit and leaves at least <FreePages> pages of EMS memory for other EMS users. <FreePages> must meet the same constraints described for the constant <ToLetFreePages> previously. <InitEMSHeap> offers the advantage that <FreePages> can be computed at runtime rather than at compile time.

After a successful call to <InitEMSHeap>, the boolean variable <EMSHeapInitialized> is set to True.

Calling this routine when <DoManualInitEMSHeap> has a value of False generates a call to the hard error function with an error code of 70. Calling this routine twice without an intervening call to <ExitEMSHeap> generates a hard error of 75.

If you are using the EMSHEAP unit to supply EMS memory to the FILER unit, be sure to call <InitEMSHeap> before calling <BTInitIsam>.

Examples

```
if DoManualInitEMSHeap then  
  InitEMSHeap(0);
```

Initializes the EMSHEAP unit to use all available EMS memory; no EMS pages will be available to other processes or units.

```
if DoManualInitEMSHeap then begin  
  assign(F, 'MYPROG.OVR');  
  reset(F, 1);  
  InitEMSHeap((FileSize(F)+16383) div 16384);  
  close(F);  
end;
```

Initializes the EMSHEAP unit to use all EMS space except for enough to load the program's overlay file into EMS.

See Also

ExitEMSHeap

Declaration

```
function MapEMSPtr(EPtr : EMSPointer) : Pointer;
```

Purpose

Maps the memory block corresponding to the logical pointer <EPtr> and returns a physical pointer to the data.

Description

This function is central to the functionality of the EMSHEAP unit. All pointer values of type <EMSPointer> returned by <GetEMSMem> may not be used directly, but must first be passed as the <EPtr> parameter to <MapEMSPtr>. <MapEMSPtr> returns the physical address of the data after it has been mapped into the EMS page frame.

Because <MapEMSPtr> returns an untyped pointer, the value must generally be typecast to a pointer of the desired type.

The pointer returned by <MapEMSPtr> has a limited lifetime; that is, further calls to <MapEMSPtr> will invalidate the pointers returned by prior calls because the EMS page frame is remapped. See the section entitled "Issues When Using EMSHEAP" above for complete information.

Example

See "Issues When Using EMSHEAP" above.

See Also

GetEMSMem

RestoreEMSCtxt / SaveEMSCtxt

Declaration

```
procedure RestoreEMSCtxt(HandleInd : Byte);
```

Purpose

Restores a previously saved EMS context.

Description

<HandleInd> is a value previously returned by <SaveEMSCtxt>.

<RestoreEMSCtxt> restores the page frame mapping that was active at the time of the call to <SaveEMSCtxt>.

Example

See "Issues When Using EMSHEAP" above.

See Also

SaveEMSCtxt

Declaration

```
function SaveEMSCtxt : Byte;
```

Purpose

Saves the current state of the EMS page frame and returns a handle index.

Description

Call <SaveEMSCtxt> when you'd like to save the page frame mapping at a particular time in order to avoid disturbing other users of EMS. <SaveEMSCtxt> returns a handle index (not a true EMS handle) that identifies the particular context to restore when <RestoreEMSCtxt> is called later. The returned value will always be between 1 and <HandlesToUseForAlloc>.

If all handles are already in use when <SaveEMSCtxt> is called, it generates a hard error of code 111. If the hard error function does not return True to halt the program, <SaveEMSCtxt> will return a handle of 255. To avoid this error, increase the constant <HandlesToUseForAlloc>, which defaults to 8.

Example

See "Issues When Using EMSHEAP" above.

See Also

RestoreEMSCtxt

B. VREC: Variable Length Records

The VREC unit uses a clever method to extend B-Tree Filer for variable length records. Such records become extremely important when a database requires memo fields, since allocating record space for the longest possible memo can waste lots of disk storage fast.

To provide support for variable length records, you first Use the VREC unit in your program, following the FILER unit on which it depends. Then, instead of calling FILER's record access routines (e.g., <BTAddRec>, <BTGetRec>, <BTPutRec>), you call the analogous routines from VREC. FILER's routines for managing fileblocks and keys are used without change. VREC's record access routines are called almost like those in FILER, with the primary exception of an added <Len> parameter, which specifies or returns the length of the variable record.

There is one other important difference. Whereas we have previously recommended that each data record reserve the first four bytes for B-Tree Filer's use, this becomes a requirement for variable length records. Otherwise it is nearly impossible to reconstruct a damaged data file.

Here is VREC's clever idea: each variable length record is composed of a series of shorter fixed length "sections". The VREC unit subdivides and recomposes a varying record from the sections. FILER manages the fixed length sections as usual.

To make this work, we must fool FILER a little bit. Specifically, the record length (<DatSLen>) that we pass to <BTCreateFileBlock> is the length of the fixed section rather than the varying length of the record itself.

How should we choose the section length? The following rules should be applied:

- the section length should be kept as small as is consistent with the remaining rules in order to conserve disk space
- the section length should typically be a multiple of 2 (64, 128, 256, 512, 1024) because DOS disk access is faster for those block sizes. An especially good block size is 512 bytes since this is the standard disk sector size
- the largest record shouldn't be much larger than about 8 sections. While this isn't a firm limit, performance will suffer if each record must be pieced together from too many sections
- a section shouldn't be much larger than the smallest version of the record
- the section length should account for the fact that VREC uses 6 bytes of the first section and 7 bytes of each subsequent section for management of the record

A contrived example helps to make these rules clear. Suppose we have an application with a fixed minimum record size of 114 bytes (deletion tag, name, company, address, phone, customer number, etc.). We have notes to add to the end of the record, ranging from 1 to 250 bytes in length (thus fitting neatly into a Turbo string). Hence, the largest record size is 364 bytes after accounting for a string length byte.

Here are two reasonable choices for section lengths:

64 byte sections

minimum number of sections: 2 (completely full: $114+1+6+7=128$)

maximum number of sections: 7 (36 bytes still available)

128 byte sections

minimum number of sections: 1 (7 bytes still available)

maximum number of sections: 3 (completely full)

For this example, the 128 byte section offers probably the best tradeoff between space and speed. NETDEMO.PAS provides a working example of variable records.

When variable length records are being used, B-Tree Filer's routines <BTUsedRecs>, <BTFreeRecs>, and <BTDatRecordSize> do not return values associated with logical record counts and sizes. As you might expect, they instead return values based on the fixed section length. Nevertheless, the values returned are still useful for computing actual disk space requirements. The VREC unit provides a replacement for <BTDatRecordSize>: <BTGetVariableRecLength> returns the actual length of a specified record.

Program organization changes slightly when using VREC. The following list indicates new requirements of the variable record length facility with an asterisk.

1. Initialize the FILER unit and create a page buffer by calling <BTInitIsam>.
2. Optionally create one or more fileblocks by calling <BTCreateFileBlock>.
3. Open one or more fileblocks by calling <BTOpenFileBlock>.
- *. Allocate a section buffer by calling <BTCreateVariableRecBuffer> or <BTSetVariableRecBuffer>.
4. Use the B-Tree Filer procedures and functions on the opened fileblocks.
5. Close opened fileblocks with <BTCloseFileBlock>.
- *. Dispose of the section buffer by calling <BTReleaseVariableRecBuffer>.
6. Dispose of the page buffer and shut down the FILER unit by calling <BTExitIsam>.

The section buffer is used to temporarily store each section while building a complete record. Only one buffer may be created at a time, no matter how many variable length fileblocks are opened. The size of the buffer must be as large as the largest section in any open fileblock that uses VREC services. Note that you may allocate the buffer by calling <BTSetVariableRecBuffer> prior to step 3, but <BCreateVariableRecbuffer> may be called only after fileblocks are open.

The following list contrasts the use of fixed and variable length records in terms of the procedures called by a program.

Opening a Fileblock

When calling <BCreateFileBlock> to create a new fileblock of variable length records, remember to specify the *section length* rather than the overall record length. The section length is stored in the data file header thereafter, so that <BOpenFileBlock> can be called as usual. Also remember to call <BCreateVariableRecBuffer> or <BTSetVariableRecBuffer> exactly one time, before calling any of the other routines in the VREC unit.

Data File Operations

When adding, deleting, or updating records, use the routines exported by the VREC unit rather than those from FILER. The following table indicates the correspondence:

Fixed Length -----	Variable Length -----
BTAddRec	BTAddVariableRec
BTDeleteRec	BTDeleteVariableRec
BTGetRec	BTGetVariableRec
BTPutRec	BTPutVariableRec

Note that the variable length routines work in single-user or network mode. Unlike fixed length records, it is not possible to read and write variable length records in spite of locks placed by other workstations. Performing reads and writes in spite of locks is inherently unreliable; in the case of fixed length records, the data may be out of date by the time the information is used. In the case of variable length records, the VREC unit might be unable to reconstruct the chain of sections; therefore the operation is not allowed at all.

Also, <BTGetVariableRec>, <BTAddVariableRec>, and <BTPutVariableRec> require an additional parameter to specify or return the length of the record in bytes.

Remember that <BTUsedRecs> returns the number of *sections* in use within a variable record length fileblock. There is no immediate way to obtain the actual number of records in use. See the <BTUsedKeys> procedure in Chapter 5 for the next best way.

Index File Operations

The use of keys is exactly the same for variable length records as for normal fixed length records. All of the same routines from the FILER unit are used identically.

Browsing a Fileblock

The BROWSER unit (described in Section 6.C) allows browsing records of either fixed or varying length. Remember to set the <Browse> function's <VarRec> parameter to True for variable length records and False otherwise.

Rebuilding or Reorganizing a Fileblock

Special versions of REBUILD and REORG are provided for variable length records. Be sure to use the VREBUILD and VREORG units in this case.

Variable Length Record Example

Let's assume that we want to add a variable length memo field to the example given in Section 4.C. The memo will be stored in a field of the following type:

```
type
  MemoFieldArray = array[1..20] of string[70];
```

After adding the new field, the record type for this application becomes:

```
type
  PersonDefVar =
    record
      Del : LongInt;
      FirstName : String[20];
      LastName : String[25];
      Street : String[30];
      City : String[30];
      State : String[2];
      ZipCode : String[9];
      Telephone : String[15];
      Age : Integer;
      Memo : MemoFieldArray;
    end;
```

The Memo field is initialized to zero length strings when it is empty. Strings are stored starting in element 1 of the MemoFieldArray when the field is used. The size of the fixed portion of the PersonDefVar is computed by

```
SizeOf(PersonDefVar)-SizeOf(MemoFieldArray)
```

This specifies how many bytes we'll need to store if the Memo field is empty. In this case the fixed portion uses 144 bytes. The largest Memo is 20*71 bytes, or 1420 bytes. These numbers lead us to choose a section length of 256 bytes. In the CreateFile example routine we replace

```
BTCreateFileBlock('TEST', SizeOf(PersonDef), 3, IID);
```

with

```
BTCreateFileBlock('TEST', 256, 3, IID);
```

A call to allocate the section buffer must also be added at the end of the OpenFile routine:

```
if not BTCreateVariableRecBuffer(PF) then begin
  OpenFile := False;
  {Error handling}
end;
```

Now we'll write a little function that returns the number of bytes to store for a given variable of type PersonDefVar, depending on how much of the Memo field is used:

```
function LenOfData(var P : PersonDefVar) : Word;
var
  LastUsed : Word;
begin
  LastUsed := 20;
  while (LastUsed > 0) and (P.Memo[LastUsed] = '') do
    dec(LastUsed);
  LenOfData := SizeOf(PersonDefVar)-SizeOf(MemoFieldArray)+71*LastUsed;
end;
```

As you can see, this routine scans from the last element in the array of strings to find a non-blank string. Then it computes the length of the overall record based on the number of strings that are used.

Given the LenOfData function, we modify the example AddRecord routine by replacing calls to

```
BTAddRec(PF, RefNr, P)
```

with

```
BTAddVariableRec(PF, RefNr, P, LenOfData(P));
```

Similarly, in the example for modifying a record, replace

```
BTPutRec(PF, RefNr, P):
```

with

```
BTPutVariableRec(PF, RefNr, P, LenOfData(P));
```

Also substitute <BTDeleteVariableRec> for <BTDeleteRec>. No parameter changes are required for this routine.

<BTGetVariableRec> returns a length word, so you must declare a new local variable of that type when you substitute the variable length routine for <BTGetRec>. When you need to determine how much of the Memo field has actually been returned by a call to <BTGetVariableRec>, you can use the following approach.

```
function NumberOfMemoLines(Len : Word) : Word;
begin
  NumberOfMemoLines := (Len-SizeOf(PersonDefVar)+SizeOf(MemoFieldArray))
                      div 71;
end;
...
var
  Len : Word
  I : Word;
...
  BTGetVariableRec(PF, RefNr, P, Len);
  for I := 1 to NumberOfMemoLines(Len) do
    Writeln(P.Memo[I]);
```

If you have an existing fixed length data file for this example, you need to convert it to variable length format before you can use it again. See Section 6.F for more information about performing the conversion.

The NETDEMO example program shows a different technique for managing a memo field. NETDEMO's approach offers somewhat reduced disk space usage and is also compatible with the memo editors of Turbo Professional and Object Professional. See NETDEMO.PAS for details.

Variable Length Record File Format

VREC splits each variable length record into sections to store on disk. It then rejoins the sections before returning the record to you in one piece. To do this, it adds a small amount of information to each section.

Let's assume a section length of 100 bytes to illustrate how VREC works. The first section of such a variable length record would include the following information:

Bytes 00..93	data from the user record
Bytes 94..95	bytes of user data in this section
Bytes 96..99	reference number of next section

Succeeding sections of the record, if any, would include the following information:

Bytes 00..00	constant value equaling 1
Bytes 01..93	data from the user record
Bytes 94..95	bytes of user data in this section
Bytes 96..99	reference number of next section

The constant byte of 1 in follow-on sections is used by the VREBUILD unit to help it find the start of each variable record (which must always begin with four zero bytes). In the last section of a variable length record, the last four bytes are zero, which terminates the chain of sections. Note that 6 bytes of the first section, and 7 bytes of succeeding sections, are used for VREC overhead.

When VREC adds a record, it stores the sections in what might seem to be reverse order. For example, suppose that you're starting with a freshly created fileblock and you're adding a record that will consume three sections. VREC will store the last third of the user record in section 1 of the fileblock, the middle third in section 2, and the starting third of the fileblock in section 3. The reference number returned by <BTAddVariableRec> will be 3. VREC must use this order internally to obtain the reference numbers needed to chain the sections together.

Fileblock Maintenance with Variable Length Records

If a fileblock was not properly closed by calling <BTCloseFileBlock>, the next attempt to open it with <BTOpenFileBlock> will return the error code 10010. This occurs when buffered data has not been written to disk.

The procedure <RebuildVFileBlock> from the VREBUILD unit is used to rebuild a variable length fileblock. It will construct new data and index files from the original data file. This is possible only if each record has reserved four initial bytes for a deletion marker. <RebuildVFileBlock> has a tougher job than the similar routine for fixed length records because it must reconnect the sections of each variable record. If the data file becomes corrupted and no backup file is available, data records following the first corrupted one may be lost. It is therefore especially important when using variable length record fileblocks to implement a good backup strategy.

VREC Identifiers

This section describes all identifiers exported by the VREC unit.

Constants

`MaxVariableRecLength = $FFF0;`

The maximum number of bytes in one variable length record, regardless of how many sections it is divided into.

Variables

`IsamVRecBufSize : Word;`

The current size of the section buffer. It is set to zero by VREC's initialization block and updated whenever you call `<BCreateVariableRecBuffer>` or `<BTSetVariableRecBuffer>`. `<IsamVRecBufSize>` must be at least as large as the largest section of any open fileblock that VREC is managing. No error will be detected if it is not, but VREC routines will then overwrite other sections of the heap, leading to unpredictable consequences. You can check `<IsamVRecBufSize>` and, if it is too small, call `<BReleaseVariableRecBuffer>` and `<BTSetVariableRecBuffer>` to allocate a larger buffer.

BTAddVariableRec

Declaration

```
procedure BTAddVariableRec(IFBPtr : IsamFileBlockPtr;  
                           var RefNr : LongInt;  
                           var Source;  
                           Len : Word);
```

Purpose

Add a variable length record to the data file.

Description

The data record is added to the end of the data file if the file has no deleted sections; otherwise one or more "holes" created by <BTDeleteVariableRec> or <BTPutVariableRec> are filled. All future references to this data record are carried out by referring to the LongInt number <RefNr> returned by the routine.

<Len> specifies the actual number of bytes in the data structure <Source>. The length must be independently determined by a method similar to that described in the introduction to this section. The same <Len> will be returned when the record is later retrieved using <BTGetVariableRec>.

A network fileblock must be locked for exclusive write access (level 1 or 2) before <BTAddVariableRec> can be called successfully.

Example

See the example at the start of this section.

See Also

BTDeleteVariableRec

BTPutVariableRec

BTCreateVariableRecBuffer

Declaration

```
function BTCreateVariableRecBuffer(IFBPtr : IsamFileBlockPtr) : Boolean;
```

Purpose

Allocate section buffer based on specified fileblock.

Description

This procedure must be called after the fileblock passed as a parameter has been opened, and before calling any procedures that will use variable length data records. An internal buffer is allocated from the heap to handle the variable length data records. The buffer's size matches the value of <DatSLen> originally used to create the fileblock (that is, the section length). If more than one variable length fileblock is to be opened simultaneously, the <IFBPtr> passed to <CreateVariableRecBuffer> must point to the fileblock with the largest value of <DatSLen>.

The buffer may be allocated only once. By calling this function with the largest data record's fileblock, a buffer large enough to handle all possible variable length data records is created. Using a buffer that is too small will lead to unpredictable consequences.

If you have multiple variable length fileblocks it may be easier to call <BTSetVariableRecBuffer> instead. With that routine you specify the desired buffer size directly instead of passing a fileblock variable.

Example

```
BTOpenFileBlock(PF, 'ADDRESS', False, False, False, False);
if not IsamOK then begin
  {Error handling}
end else begin
  if not BTCreateVariableRecBuffer(PF) then begin
    {Error handling}
  end;
end;
```

Opens an existing variable record fileblock and allocates a section buffer just large enough for it.

BTDeleteVariableRec

Declaration

```
procedure BTDeleteVariableRec(IFBPtr : IsamFileBlockPtr;  
                             RefNr : LongInt);
```

Purpose

Delete a variable length record from data file.

Description

The deleted sections are really just marked as free, so that future additions with <BTAddVariableRec> can reuse the space. <RefNr> specifies the data reference number as returned by <BTAddVariableRec>, or later by a call to <BTFindKey> or other key search routines.

Only existing data records may be deleted. Deleting an already deleted record corrupts an internal linked list of deleted data records and therefore has unpredictable results. When a record is deleted, B-Tree Filer sets the first four bytes of each section to a non-zero value.

In a network application, this procedure may be called successfully only when the network fileblock has been locked at degree 1 or 2.

Example

```
BTFindKey(PF, 1, RefNr, UserKeySt);  
if IsamOK then  
  BTGetVariableRec(PF, RefNr, Len);  
if IsamOK then begin  
  {Prompt user whether to delete}  
  ...  
  BTDeleteVariableRec(PF, RefNr);  
  if not IsamOK then begin  
    {Error handling}  
  end;  
end;
```

Uses the index system to find a particular record, then offers the user an opportunity to confirm the deletion. Note that the call to <BTGetVariableRec> is only needed to provide information for prompting the user; <BTDeleteVariableRec> does not use it.

See Also

BTAddVariableRec

BTGetVariableRec

Declaration

```
procedure BTGetVariableRec(IFBPtr : IsamFileBlockPtr; RefNr : LongInt;  
    var Destination; var Len : Word);
```

Purpose

Read a variable length record with reference <RefNr>.

Description

<RefNr> is normally obtained from a prior index operation. The actual length of the data record is returned in the parameter <Len>. The record is pieced back together from the chain of sections.

Since <Destination> is untyped, the compiler will not detect an invalid variable passed in this position. Assure that the variable passed is large enough to hold the longest data record.

Example

See the example at the beginning of this section.

See Also

BTGetVariableRecLength

BTGetVariableRecPart

BTGetVariableRecLength

Declaration

```
procedure BTGetVariableRecLength(IFBPtr : IsamFileBlockPtr; RefNr : LongInt;  
                                var Len : Word);
```

Purpose

Return the length of record <RefNr>.

Description

<RefNr> is normally obtained from a prior index operation.

<BTGetVariableRecLength> takes just as much time as reading the entire record, so it should be used only when a buffer size must be determined before actually reading the record.

Example

```
var  
  BufPtr : Pointer;  
  Len : Word;  
const  
  BufSize : Word = 0;  
...  
  BTGetVariableRecLength(PF, RefNr, Len);  
  if Len > BufSize then begin  
    if BufSize <> 0 then  
      FreeMem(BufPtr, BufSize);  
    if MaxAvail >= Len then begin  
      BufSize := Len;  
      GetMem(BufPtr, BufSize);  
    end;  
  end;  
  BTGetVariableRec(PF, RefNr, BufPtr^, Len);
```

Demonstrates the technique to dynamically allocate a buffer that is as large as the largest variable record. Real code needs more error handling.

See Also

BTGetVariableRec

BTGetVariableRecPart

BTGetVariableRecPart

Declaration

```
procedure BTGetVariableRecPart(IFBPtr : IsamFileBlockPtr;  
                               RefNr : LongInt;  
                               var Destination;  
                               var Len : Word);
```

Purpose

Reads part of a variable length record.

Description

<RefNr> is normally obtained from a prior index operation. Unlike <BTGetVariableRecord>, this routine returns just the first <Len> bytes of the variable length record. This is intended primarily to allow reading the fixed portion of a record that has a variable length memo field at its end.

If there are fewer than <Len> bytes in the record, only those that exist will be returned in <Destination>. <BTGetVariableRecPart> returns the actual number of bytes it could read in the <Len> parameter.

Example

```
var  
    Len : Word;  
...  
Len := SizeOf(PersonDefVar)-SizeOf(MemoField);  
BTGetVariableRecPart(PF, RefNr, P, Len);
```

Reads just the fixed portion of the PersonDefVar record described in the introduction.

See Also

BTGetVariableRec

BTGetVariableRecLength

BTGetVRecPartReadOnly

Declaration

```
procedure BTGetVRecPartReadOnly(IFBPtr : IsamFileBlockPtr;  
                                RefNr : LongInt;  
                                var Destination;  
                                var Len : Word);
```

Purpose

Reads part of a variable length record despite a record lock placed by another workstation.

Description

This routine works just like <BTGetVariableRecPart> except that it will read a record that has been locked by another workstation.

In contrast to fixed length records, a variable length record cannot be read unless a record lock, fileblock lock, or readlock has been placed by the calling workstation. This constraint is due to the fact that variable length records are composed of a chain of fixed length records, which might change in the midst of a read if not lock were in place. If no lock is already in place, <BTGetVRecPartReadOnly> places a temporary readlock on the fileblock. If this readlock cannot be obtained immediately (because another station has a writelock on the fileblock), an IsamError of 10332 is generated.

As in the corresponding procedures for fixed length records, <BTGetVRecPartReadOnly> doesn't read the first four bytes of the data record if the record is locked by another workstation. The first four bytes of <Destination> are left uninitialized in this case, and <IsamError> returns the warning code 10205.

Note that if you receive the 10205 warning, you cannot immediately determine whether the corresponding record has been deleted. (Normally you would test the first four bytes for a non-zero value.) However, the following logic will guarantee that the record has not been deleted, even if warning 10205 occurs:

```
Readlock  
Find an associated key (using <BTFindKey> for example)  
Read record using <BTGetVRecPartReadOnly>  
Unlock
```

The data record is guaranteed to exist because its corresponding key exists; the key could not have been deleted in the meantime because a readlock was placed.

See Also

BTGetVariableRecPart

BTGetVRecReadOnly

BTGetVRecReadOnly

Declaration

```
procedure BTGetVRecReadOnly(IFBPtr : IsamFileBlockPtr;  
                           RefNr : LongInt;  
                           var Destination;  
                           var Len : Word);
```

Purpose

Reads a variable length record despite a record lock placed by another workstation.

Description

This routine works just like <BTGetVRecPartReadOnly> except that it returns the complete variable length record in <Destination> and the length of the record in <Len>.

See Also

BTGetVariableRec

BTGetVRecPartReadOnly

Declaration

```
procedure BTPutVariableRec(IFBPtr : IsamFileBlockPtr;  
                           RefNr : LongInt;  
                           var Source;  
                           Len : Word);
```

Purpose

Write a variable length record back to data file.

Description

<RefNr> is normally obtained from a prior index operation. <Len> contains the actual length of the data record, which may differ from the original value. Even when <Len> is longer than the original value, the data reference number for the record remains unchanged after the call to <BTPutVariableRec>. Whenever the record uses more than one section, <BTPutVariableRec> relocates sections other than the first one, which creates free sections that will be reused later.

Never add a new data record to the data file with <BTPutVariableRec>. Use <BTAddVariableRec> instead. <BTPutVariableRec> may be used only to write a data record back to the same location from where it was read. Indiscriminately writing into the data file with <BTPutVariableRec> will lead to unpredictable results.

In a network application, this procedure may be called successfully only when the network fileblock is locked for exclusive write access (level 1 or 2).

Example

```
BTGetVariableRec(PF, RefNr, P, Len);  
FillChar(P.Memo, SizeOf(MemoFieldArray), 0);  
BTPutVariableRec(PF, RefNr, P, LenOfData(P));
```

Reads a variable length record of the type used in the introductory example, clears the memo field, and puts the record back to the data file with its new shorter length.

See Also

BTAddVariableRec

BTReleaseVariableRecBuffer

Declaration

```
procedure BTReleaseVariableRecBuffer;
```

Purpose

Deallocates the section buffer.

Description

This procedure frees the memory allocated by <BTPCreateVariableRecBuffer> or <BTSetVariableRecBuffer>. It should be called only when all variable length record operations are complete, or perhaps when the current buffer is found to be too small and must be enlarged.

Example

```
BTOpenFileBlock(P, 'ADDRESS', False, False, False, False);
if IsamVRecBufSize < BTDatRecordSize(P) then begin
  BTReleaseVariableRecBuffer;
  if not BTPCreateVariableRecBuffer(P) then begin
    (Error handling);
  end;
end;
```

After a new fileblock is opened, this code assures that the variable record section buffer is large enough. If not, it releases the existing buffer (if any) and allocates a new larger one.

See Also

BTPCreateVariableRecBuffer

BTSetVariableRecBuffer

Declaration

```
function BTSetVariableRecBuffer(Size : Word) : Boolean;
```

Purpose

Allocate section buffer of specified size.

Description

This function is an alternative to <BTCreateVariableRecBuffer>. Instead of taking the buffer size from internal fields in a fileblock, the size is directly specified with the call. This may be more convenient in some cases.

Never use <BTCreateVariableRecBuffer> and <BTSetVariableRecBuffer> one after the other. Only one of the two may be used. The memory used by calling either of these two routines will be freed with <BTReleaseVariableRecBuffer>.

Example

```
if not BTSetVariableRecBuffer(256) then begin
  {Error handling}
end;
```

Allocates a section buffer without referring to any particular fileblocks. Remember that the size specified must be at least as large as the largest section length for any fileblock used with the VREC unit.

See Also

BTCreateVariableRecBuffer

C. BROWSER: Displaying a Fileblock

The browser is a utility used to examine a selection of data records from a fileblock. The data records are displayed on-screen using one row per record, in a format that you define. A highlight bar can be positioned to select a particular record.

The BROWSER unit is extensively reconfigurable:

- browser input comes from any fileblock, using fixed length or variable length records, in a single user or network environment
- browser output can be positioned within any desired window
- the order in which records are displayed can be based on any index of the fileblock
- low and high key values specify the range of records to display, and a filtering function can be used to select records meeting additional criteria
- formatting and display of records are handled by user-defined procedures whose addresses are passed to the browser at runtime
- commands for vertical and horizontal scrolling of the records are provided
- the keyboard can be remapped and custom exit keys can be defined
- for multi-user applications a hook allows the browser to automatically update the display whenever another workstation modifies the fileblock
- mouse and scroll bar support are available to owners of Turbo Professional or Object Professional

To use the BROWSER unit in your program, just add it to the Uses statement. BROWSER also depends on the following units: CRT, DOS, FILER, and VREC. If you're using Turbo Professional, be sure to edit BTDEFINE.INC and activate the conditional define UseTPCRT. If you're using Object Professional, be sure to activate UseOPCRT in BTDEFINE.INC. In these cases, the setting for UseMouse in TPDEFINE.INC or OPDEFINE.INC respectively determines whether the browser supports the mouse.

Note that the BROWSER unit does not fit into Object Professional's window hierarchy. If you'd like an object-oriented version of the browser, dearchive FBROWSE.LZH on the B-Tree Filer BONUS disk and read FBROWSE.DOC.

The command handling behavior of BROWSER is implemented just like it is in the Turbo Professional units TPPICK, TPENTRY, and others. A configurable command table is defined to map keystrokes to logical commands like "move to next record". Although this table is initialized to reasonable default values, an application

can modify it at runtime for special capabilities. See the section on "BROWSER Keyboard Handling", below, for more details.

For correct behavior of the browser, the first four bytes of each data record must be reserved for B-Tree Filer's use, and initialized by the application to zero whenever a record is added to the fileblock.

BROWSER Example

We'll add to the example of Section 4.C to introduce how the browser works. First, we must add the CRT (or TPCRT or OPCRT) and BROWSER units to the uses statement.

A minimum of two user-defined routines are needed to control how the browser displays each record: one to create a text string that contains the visible contents of each record, and another to write each such string to the screen at the desired location in the desired attributes. The browser calls the first routine whenever it has read a new record from the fileblock, and it calls the second whenever it updates the screen. The second routine is called more frequently; the separation of the two routines leads to better performance.

We call the first routine BuildARow. It must be global (not nested), it must be compiled with the far model, and it must have exactly the parameters specified. Here's our routine:

```
{SF+}
procedure BuildARow(var RR : RowRec; KeyNr : Integer;
                   var DatS; DatLen : Word);
  (-Return one row to the browser)
begin
  with RR, PersonDef(DatS) do begin
    if Ref <> -1 then
      Row := FirstName+ ' '+LastName+ ' '+City
    else
      {Record is locked, indicate it on screen}
      Row := '*****';
    Row := Pad(Row, 80);
  end;
end;
```

<RowRec> is a type interfaced by BROWSER. It contains three fields: IKS, the key string for a particular record; Ref, the data reference number for the record; and Row, a string that is to be initialized by BuildARow. The RR.IKS and RR.Ref fields are already set when BuildARow is called. The parameter DatS contains the actual data record and DatLen contains the number of bytes in the data record (which is particularly useful for variable length records).

When RR.Ref is set to -1, the browser was unable to read this particular record because it was locked. BuildARow should set RR.Row to a distinctive string, perhaps a series of asterisks, in this case. If RR.Row contains any other value, BuildARow should combine RR.Row, RR.IKS, and the fields of DatS in any fashion to build a display string. In our simple example we just concatenate the FirstName, LastName, and City fields. Note how it's necessary to typecast the untyped DatS parameter to the actual record type in order to access the data fields.

KeyNr is the key number the browser is using to determine the display order. You may use this parameter to select different display formats for different keys.

The second routine, DisplayARow, is responsible for writing the RowRec to the screen. Again, it must be global, compiled far, and have exactly the specified parameters, as follows:

```
{F+}
procedure DisplayARow(var RR : RowRec; KeyNr, RowNr, StartRow : Integer;
                    Highlight : Boolean; var HorizOfs : Integer);
  {-Display one row for the browser}
begin
  if Highlight then
    TextAttr := $70
  else
    TextAttr := $07;
    GotoXY(1, StartRow+RowNr-1);
    Write(RR.Row);
end;
```

All values passed to DisplayARow are initialized by the browser. RR is a <RowRec> previously set up by the BuildARow routine. Although RR is a Var parameter, DisplayARow should not change it. KeyNr is again the index number being used to determine browsing order. RowNr is the relative row number within the browse window for the particular record to be displayed, in the range 1..<NrOfRows>, where <NrOfRows> is the total browsing rows passed to the <Browse> function. StartRow is the first actual screen row used for the browser, again based on a parameter passed to <Browse>. Highlight is True if the record has the "highlight bar", i.e., is the current record. This parameter is often used to select a video attribute. HorizOfs specifies the amount of horizontal scrolling; it can often be ignored, as it is here.

As you can see, the example routine chooses a video attribute of \$70 (reverse video) when Highlight is True, or \$07 (gray on black) when it's not. It then moves the CRT unit's cursor to the left edge of the screen on the appropriate row and writes the record's previously computed string to the screen. Note that we've previously padded the row strings to the maximum screen width. If we hadn't done so, it would be DisplayARow's responsibility to clear out the end of each line.

The following example shows the variables and main code block needed to activate the browser.

```

var
  PF : IsamFileBlockPtr;
  Pages : LongInt;
  BrowseStatus : Integer;
  P : PersonDef;
  Len : Word;
  RefNr : LongInt;
  KeyStr : IsamKeyStr;
  ExitCmd : BKType;
begin
  Pages := BtInitIsam(NoNet, 40000, 0);
  if not IsamOK then begin
    {Error handling}
    Halt;
  end;
  BtOpenFileBlock(PF, 'TEST', False, False, False, False);
  if not IsamOK then begin
    {Error handling}
    Halt;
  end;

  RefNr := 1;
  KeyStr := '';
  ExitCmd := BKNone;
  BrowseStatus :=
    Browse(PF,           {Open fileblock pointer}
           False,        {Variable length records?}
           1,            {Index for browse order}
           '',           {Lowest key to display}
           #255,         {Highest key to display}
           2,            {Screen row for first row of browser}
           20,           {Number of rows for browser}
           P,            {Data record buffer for selected record}
           Len,          {Length of returned record}
           RefNr,        {Record number of selected record}
           KeyStr,       {Key string for selected record}
           ExitCmd,      {Command used to exit browser}
           Nil,          {Special task hook}
           @BuildARow,   {User routine to build each row string}
           @DisplayARow); {User routine to display each row string}

  CTrScr;
  case BrowseStatus of
    0 : Writeln('Exited on record ', RefNr, ' with exit command ', ExitCmd);
    1 : Writeln('No keys available in specified range');
    2 : Writeln('FILER error ', IsamError, ' while browsing');
  end;
  BtExitIsam;
end.

```

The first several lines of code are standard preparation: the FILER unit is initialized via <BtInitIsam> and a fileblock is opened with <BtOpenFileBlock>.

The call to `<Browse>` activates the browser. We'll discuss its parameters in detail in the reference section following. For now, note that we pass it the addresses of the `BuildARow` and `DisplayARow` routines. `<Browse>` draws a screenful of records and then takes control of the keyboard to allow the user to scroll around and make a selection.

`<Browse>` returns two status variables that the application must check. The function result `BrowseStatus` contains overall status from the browser; any value other than zero indicates that an error occurred and therefore no selection could be made. If `BrowseStatus` is zero, then `ExitCmd` contains a command number used to exit the browser. This is typically `<BKenter>` (user pressed `<Enter>`) or `<BKquit>` (user pressed `<Esc>`). Additional program-defined exit commands are possible as well. At this point, the parameters `RefNr`, `KeyStr`, `Len`, and `P` are also initialized to contain the current record values.

Record Filtering

You may not want to display all the records in a database while browsing. There are three ways to do this:

- when you call `<Browse>`, specify `<LowKey>` and `<HighKey>` values to limit the displayed records to a particular range
- build an index that contains keys for only those records you wish to display
- take advantage of the browser's record filtering hooks

The first approach should be chosen whenever a simple range selection is adequate.

The second approach can be implemented in several ways. It works best when you reserve an index slot for the "browsing index" at the time the fileblock is created, and add and delete appropriate keys for this index whenever records are added and deleted. In this way, there's no delay to generate the index when you're ready to start the browser.

Sometimes you may not be able to generate the browsing index in advance, perhaps because the selected records are based on a user-defined specification. You can still create a temporary index just before calling the browser, although your user will have to wait while that index is generated. If you've reserved an index slot for this temporary use, you can clear the index by calling `<BTDeleteAllKeys>` and then add keys for just the selected records to that index. See Chapter 5's entry for `<BTDeleteAllKeys>` for an example.

You can even generate a temporary index if you haven't reserved a slot for it when the fileblock was created. The following code fragment shows how.


```

var
  PF : IsamFileBlockPtr;
  TPF : IsamFileBlockPtr;
  IID : IsamIndDescr
  RefNr : LongInt;
  P : PersonDef;
...
{Initialize temporary index descriptor}
IID[1].KeyL := 20;
IID[1].AllowDupK := False;
{Create temporary fileblock}
BTCreateFileBlock('TEMP', SizeOf(PersonDef), 1, IID);
{Open a fileblock using the real data file and a temporary index file}
BTOpenFileBlock(TPF, 'TEST;TEMP');

{Add the keys to the temporary index}
for RefNr := 1 to BTFilen(TPF) do begin
  BTGetRec(TPF, RefNr, P, False);
  if SomeConditionIsMet(RefNr, P) then
    BTAddKey(TPF, 1, RefNr, TempBuildKey(1, P));
end;

{Browse using the temporary index}
RefNr := 1;
KeyStr := '';
ExitCmd := BKNONE;
BrowseStatus :=
  Browse(TPF,           {Open fileblock pointer}
    False,             {Variable length records?}
    1,                 {Index for browse order}
    '',                 {Lowest key to display}
    #255,               {Highest key to display}
    ...);

{Close the temporary fileblock and delete temporary files}
BTCloseFileBlock(TPF)
BTDeleteFileBlock('TEMP');

```

This technique takes advantage of the fact that you can separately specify the data and index file names for a fileblock. In this case we use the data file from the real fileblock PF, but we create a new index file that will exist only temporarily for the use of the browser. The technique must be used carefully--if at all--in a network setting; if another workstation modified the data file it wouldn't know to update the temporary index file and the browser could get dangerously out of sync with reality.

The disadvantages of the preceding techniques caused us to add record filtering hooks to the browser itself. These hooks allow you to filter the desired records in real time using any index.

Suppose you want to display records in last name order, for only those records whose zip code starts with '9'. To do so, you write a record validation function, which the browser will call in order to determine whether each record meets the

filtering requirements. The record validation function can perform whatever logical tests that it requires, including a) reading the data record associated with the key and checking other fields, or b) checking the data in related fileblocks.

Note that filtering can slow browser operations substantially. Imagine that you have a fileblock containing 100,000 records and that your filtering function accepts only 100 of these records, spread equally throughout the key sequence. In this case, the browser would need to scan 1,000 records for every record that it eventually displayed on the screen.

Use the following routines to control record filtering:

```
procedure EnableFiltering(ValidateFunc : Pointer);
  {-Enables Browser filtering. <ValidateFunc> is a pointer to a
   user-defined function that determines whether a given record
   should be displayed in the Browser}

procedure DisableFiltering;
  {-Disables Browser filtering. Has no effect if filtering is not
   enabled}

function IsFilteringEnabled : Boolean;
  {-Returns True if Browser filtering is enabled}
```

<ValidateFunc> must be the address of a far, non-nested function declared as follows:

```
{SF+}
function ValidateARecord(IFBPtr : IsamFileBlockPtr;
  KeyNr : Integer;
  Ref : LongInt;
  var KeyStr : IsamKeyStr;
  NetUsed : Boolean) : Boolean;
```

The record validation function returns True if the indicated record is to be displayed by the Browser, False if not. At the time the validation function is called, the browser has not read the data record by calling <BTGetRec>. That saves time in many applications where the filtering decision can be made simply by checking the key string (KeyStr) or the data reference number (Ref). The validation function may of course read the record or perform other tests before making its decision. NetUsed is True if IFBPtr is a network fileblock; this parameter remains for historical reasons (even though network status can be determined from IFBPtr).

SIMPDEMO and NETDEMO provide working examples of record filtering. The <F6> command lets you specify fields of the address record to be used as filter masks; once enabled, the browser displays only those records consistent with the filter masks.

Automatic Screen Refresh

In a network environment, several users may be browsing and modifying a fileblock at the same time. The BROWSE unit is designed to deal with this situation automatically. By default, changes made on another station do not appear on the screen until a keystroke that reloads the page buffers is executed. For example, pressing <PgDn> causes the browser to read a new set of keys and records from disk; any changes made by other stations will appear at that time.

The browser also includes a "refresh function" hook which allows changes to appear in real time, without waiting for a special keystroke. The refresh function's purpose is to detect that another workstation has modified the fileblock and signal the browser to update the screen. When activated, the refresh function is called just prior to getting each keyboard command while the browser is active. A refresh function is activated by pointing BROWSER's global typed constant <RefreshFunc> to a global, far routine declared as follows:

```
{SF+}  
function RefreshFunction(IFBPtr : IsamFileBlockPtr;  
                        KeyNo : Integer) : Boolean;
```

When no refresh function is desired, RefreshFunc should be set equal to Nil, as it is by default. The Refresh function should return True if a screen refresh is required, and False if not.

BROWSER interfaces two predefined refresh functions, <RefreshAtEachCommand> and <RefreshPeriodically>. The first signals for a screen refresh if 1) no keystrokes are pending and 2) the fileblock has been modified since the last refresh. The second checks every <RefreshPeriod> clock ticks to see whether the fileblock has been modified. If a key is pressed prior to detecting a modification, <RefreshPeriodically> exits with a False result; otherwise it returns True. The global typed constant <RefreshPeriod> defaults to 90 clock ticks (about 5 seconds). <RefreshPeriodically> usually generates less network traffic because it checks for a modification less frequently; however, screen updates won't occur as quickly after other stations modify the data.

The two predefined refresh functions will serve in most applications. However, if an application needs to perform other tasks while waiting for a keypress in the browser, it should use the provided functions as a model for routines that perform the additional tasks as well.

SIMPDEMO and NETDEMO provide working examples of the refresh function when dealing with network fileblocks. When these demos are used in a NetWare environment, they activate a special refresh function that uses NetWare semaphore services to detect fileblock modification with very low overhead.

Mouse Support

In order to use the browser's mouse support, you must have either Turbo Professional or Object Professional and you must activate either UseTPCRT or UseOPCRT in BTDEFINE.INC. Additionally, UseMouse must be defined in TPDEFINE.INC or OPDEFINE.INC. (BROWSER uses the TPCRT and TPMOUSE units from Turbo Professional, or OPCRT and OPMOUSE from Object Professional. You can still use the browser *without* mouse support if you just have Borland's CRT unit.) If a mouse is present, call <EnableBrowseMouse> to enable browser mouse support, and <DisableBrowseMouse> if needed to disable it later.

When the mouse is enabled, clicking over a particular record will highlight it; clicking again will select that record (causing <Browse> to exit the same as if <Enter> were pressed). The browser will optionally display a vertical scroll bar, including a slider that indicates the approximate relative position of the highlighted record among all the keys of the browser index. The scroll bar works like one in a Turbo Professional pick list: clicking on the bar scrolls the display to the indicated relative location; clicking on the up arrow of the scroll bar moves the highlight bar up by one row or one page; clicking on the down arrow moves the bar down in the same fashion. If the mouse is outside of the browser screen region when the mouse is clicked, an optional user routine is called.

The following typed constants control the browser's mouse support:

```
BrowseMouseEnabled : Boolean = False;  
    (True if mouse is enabled)  
  
ScrollBarAttr : Byte = $07;  
    (Video attribute for scroll bar)  
  
SliderAttr : Byte = $0F;  
    (Video attribute for slider)  
  
MouseUpMark : Char = #24;  
    (Arrow character at top of scroll bar)  
  
MouseDownMark : Char = #25;  
    (Arrow character at bottom of bar)  
  
ScrollMark : Char = #178;  
    (Character for slider)  
  
ScrollVertChar : Char = #176;  
    (Character for scroll bar background)  
  
UserMousePtr : Pointer = Nil;  
    (Address of mouse user routine)  
  
BrowseMousePage : Boolean = False;  
    (True to scroll by one page per click)
```

```

AutoScaleMouse : Boolean = True;
  {Adjust bar for LowKey and HighKey?}

UseScrollBar : Boolean = True;
  {Display scroll bar?}

ScrollBarAutoSize : Boolean = True;
  {Match bar to window height?}

ScrollBarUp : Byte = 1;
  {Relative location of bar's up arrow}

ScrollBarHt : Byte = 18;
  {Height of bar, excluding the arrows}

ScrollBarCol : Byte = 80;
  {Absolute column for scroll bar}

MouseX1 : Byte = 1;
  {Left margin for mouse select}

MouseX2 : Byte = 79;
  {Right margin for mouse select}

```

<ScrollBarAttr> and <SliderAttr> are the video attributes used when displaying a scroll bar. <MouseUpMark> and <MouseDownMark> are the characters used for the up and down arrows of the scroll bar. <ScrollMark> is the character used for the "slider". <ScrollVertChar> forms the background for the scrollbar.

<UserMousePtr>, if not Nil, points to a routine of the following form (a global routine compiled under the far model):

```

{$F+}
procedure MouseHotSpotHandler(X, Y : Byte; var Cmd : BKtype);
begin
  ...
end;
{$F-}

```

<X> and <Y> are the absolute coordinates of the mouse cursor at the time the mouse was clicked somewhere outside the browse window. <Cmd> is the command to be executed next, if any, based on the position of the mouse. The application should set Cmd to <BKNone> if no browser command will be executed as a result of the mouse click.

<BrowseMousePage>, if True, indicates that clicking on the up/down arrows should scroll the display by a full page.

<AutoScaleMouse> indicates whether the range of the scroll bar should extend from <LowKey> to <HighKey> (the default), or from the lowest possible key to the highest possible key. In the latter case, if <LowKey> and <HighKey> specify a

constrained range of key values, then the slider will be constrained to fall within a corresponding range of the scroll bar.

<UseScrollBar> indicates whether or not a scroll bar is desired; it is forced to False if a mouse is not present. If True, <ScrollBarAutoSize> indicates that the size and vertical position of the scroll bar should be calculated automatically based on the height and location of the browse window. If False, <ScrollBarUp> and <ScrollBarHt> indicate the top row and the height of the scroll bar. <ScrollBarCol> is the absolute screen column in which the scroll bar should appear.

Keyboard Handling

The browser offers a number of hooks for customization of keyboard operations. These features are modeled after those first introduced with Turbo Professional 5.0, so if you're familiar with keyboard handling in TPENTRY or TPEDIT, the browser's features will be easy to use. If not, read on.

There are three kinds of hooks that fall into the category of keyboard handling. The first is the "installable keyboard hook". BROWSER defines a set of logical commands that are tied to a particular set of keystrokes, and it stores the links between commands and keystrokes in an array that can be easily modified. The second is the "background task hook". BROWSER allows the programmer to specify a function that returns a keystroke, and that function is free to perform whatever tasks it wishes while waiting for a key to be pressed. The third is the "context-sensitive help hook". This hook facilitates the creation of context-sensitive help systems by providing a means of invoking a programmer-defined help routine whenever a particular command is issued.

BROWSER's installable keyboard hooks are accessed by using the function <AddBrowseCommand>. This function reads and modifies a typed constant array of bytes that holds the mapping between logical commands and keystroke sequences. BROWSER's array is called <BrowseKeySet>, and it can hold up to 201 bytes of mapping information. By default, it is configured to support all the obvious cursor keys for paging through a list of records, and also the traditional WordStar "magic diamond". See <AddBrowseCommand> for instructions on how to modify this command list while your program runs. For substantial modifications to the list, you have two other options. The first is simply to modify BROWSER's source code to specify the desired keys. The second is to write an installation program that finds the key list by means of an identification string conveniently placed just before the table. How to write such an installation program won't be discussed here; there are examples provided with Borland's Editor Toolbox and with Turbo Professional (TPKEYS, on the BONUS disk).

Background task hooks allow a program to get some work done while waiting for keyboard input. The most common application of this technique is to display the current time. In a network application, there may be other needs, such as polling for messages from other workstations or attempting to obtain file locks. BROWSER's background task hook makes this easy to do.

To define a background task, you must declare a function of the following form:

```
{F+}
function MyReadKey : Word;
begin
  while not KeyPressed do begin
    inline($CD/$28); {Give TSR's a chance to pop up}
    (** Perform background task here **)
  end;
  MyReadKey := BrowseReadKey;
end;
{F-}
```

The function must be compiled with the far model and must be global (not nested within any other routines). The KeyPressed loop polls for waiting keystrokes and provides continuing opportunities for a background task to execute. The inline statement is optional, but it is sometimes necessary to allow TSR's to pop up while you're waiting for keyboard input. Of course, the background task itself must not take too long, or keyboard response will become sluggish.

To activate the background task, assign the address of your function to BROWSER's variable <BrowseKeyPtr>, like so:

```
BrowseKeyPtr := @MyReadKey;
```

(If you use the automatic screen refresh function <RefreshPeriodically> described previously, you'll find that the background task facilities described here are effectively disabled. That's because the checking performed for automatic screen refresh is itself a background task. Although the routine you specify with <BrowseKeyPtr> will still be called, it won't be called until a keystroke is already pending and hence your background task won't get any time. If you need to use both <RefreshPeriodically> and your own background tasks in one program, you must write an automatic screen refresh function that combines both functions.)

Activating a context sensitive help hook involves a similar process. When the <F1> key is pressed, BROWSER checks to see if the <BrowseHelpPtr> pointer is Nil. If it is, the <F1> key is ignored. If not, the routine at the specified address is called. (Of course, the <F1> key mapping can be changed using the installable keyboard hook.)

The routine called when the help key is pressed must take a very specific form:

```

{$F+}
procedure HelpRoutine(UnitCode : Byte; IdPtr : Pointer; HelpIndex : Word);
begin
end;
{$F-}

```

Again, the routine must be compiled with the far model, and it must be global. The parameters must match those shown in both type and order. The general form used here is just like that in Turbo Professional, which defines such a routine for a number of units, including those for line editing, data entry screens, and menus. The parameters allow the help routine to determine what unit called it, so that it can provide help in an appropriate form.

When such a routine is called from BROWSER, the parameters take on a special meaning. The first parameter will always have the value <HelpForBrowse>, which is a constant declared by BROWSER. This indicates that the call for help came from the database browser. The second parameter, <IdPtr>, will be set equal to the fileblock pointer being browsed. The third parameter, <HelpIndex>, will be assigned the value <BrowseHelpIndex>, which is a variable that the application should set just prior to calling <Browse>. The help hook routine can then decide what help is appropriate to display based on these parameters. Displaying the help is the application's responsibility; if you have Turbo Professional, the TPHELP unit is ideal for the task.

To activate the help hook, assign the address of your procedure to BROWSER's variable <BrowseHelpPtr>, like so:

```

BrowseHelpPtr := @HelpRoutine;

```


BROWSER Identifiers

This section describes all identifiers exported by the BROWSER unit.

Constants

A number of typed constants are used for mouse control. These typed constants are declared only if the UseMouse conditional is defined and other conditions are met. See "Mouse Support" earlier in this section for a description of these constants.

```
BKnone      = 00; {Not a command}
BKchar      = 01; {Regular character--not a command}
BKenter     = 02; {Select}
BKquit      = 03; {Escape}
BKfirstRec  = 04; {Cursor to first record}
BKlastRec   = 05; {Cursor to last record}
BKleft      = 06; {Cursor left one column}
BKright     = 07; {Cursor right one column}
BKup        = 08; {Cursor up one row}
BKdown      = 09; {Cursor down one row}
BKpageUp    = 10; {Cursor up one page}
BKpageDown  = 11; {Cursor down one page}
BKplus      = 12; {Reread current record}
BKhelp      = 13; {Invoke help routine}
BKredraw    = 14; {Redraw the browse screen}
BKprobe     = 15; {Signals a mouse event}
BKrowEnd    = 16; {Go to end of row}
BKrowBegin  = 17; {Go to start of row}
BKtask0     = 18; {User-defined task commands}
...
BKtask9     = 27;
BKuser0     = 28; {User-defined exit commands}
...
BKuser9     = 37;
```

These are the codes for each of BROWSER's cursor movement and selection commands. Most of them are self-explanatory. The <BKhelp> command is activated whenever <F1> is pressed. If a user-defined help routine has been specified, it is called at this time.

<BKredraw> (which is not mapped to any keystroke) is used internally for mouse support. <BKprobe> is triggered whenever the left mouse button (by default) is pressed. BROWSER then classifies the mouse position and acts accordingly.

By default, neither <BKrowEnd> nor <BKrowBegin> is mapped to a keystroke sequence. When <BKrowEnd> is executed, the browser sets its internal horizontal scrolling offset to 32767 and passes this value to the user DisplayARow routine, which can interpret the value to mean "scroll horizontally as far to the right as possible". The DisplayARow routine must then clip <HorizOfs> to a realistic value so that subsequent <BKleft> and <BKright> commands work correctly.

<BKrowBegin> resets the horizontal offset to zero. NETDEMO and SIMPDEMO

demonstrate the use of these commands, mapping them to the <End> and <Home> keys respectively.

<BKtask0>..<>BKtask9> may be assigned to keystroke sequences that when pressed cause BROWSER to call the user-defined special task routine. Keystrokes assigned to <BKuser0>..<>BKuser9> will cause the browser to exit. The application program can then inspect the returned parameter <ExitKey> to determine what action to take.

```
BrowseKeyMax = 200;  
BrowseKeyID : String[17] = 'browser key array';  
BrowseKeySet : array[0..BrowseKeyMax] of Byte = (...);
```

Used to identify and manage the keystroke to command mapping table. See "Keyboard Handling" earlier in this section for details.

```
($IFDEF UseTPCRT)  
HelpForBrowse = Tpcrt.HelpForXXXX2; {= 7}  
($ELSE)  
($IFDEF UseOPCRT)  
HelpForBrowse = 99;  
($ELSE)  
HelpForBrowse = 7;  
($ENDIF)  
($ENDIF)
```

Defines the unit index passed to a context sensitive help routine. See "Keyboard Handling" earlier in this section for details.

```
MaxCols = 128;
```

Specifies the maximum width of a one-line formatted record to be displayed by the browser. This may be larger than the width of the physical screen provided that horizontal scrolling is supported by the user-defined routines. It cannot be greater than 255. This constant, in combination with <MaxRows>, determines the dominant stack space usage of the <Browse> function. With default values, <Browse> uses somewhat less than 4000 bytes of stack space.

```
MaxRows = 20;
```

Specifies the tallest window allowed by the browser. If the actual window (specified in a call to <Browse>) is taller, BROWSER will limit the height to <MaxRows>. This constant determines the size of a variable local to the <Browse> function.

```
MinRows = 4;
```

Specifies the smallest window height allowed by the browser. If the actual window (specified in a call to <Browse>) is shorter, BROWSER will increase the height to <MinRows>. BROWSER isn't designed to work with smaller values for <MinRows>, so the only option is to make it larger.

NoNetMode : Boolean = False;

In a network environment, the browser must read from the disk more often than for a single user application. For example, another workstation might delete the first record currently displayed by the browser, requiring a complete screen update. When NoNetMode is False, the browser rebuilds each display page from scratch after most commands. When it is True, the browser assumes that no records are deleted or modified until the browser exits; in that case, browser performance will be much snappier.

ReadDataRecord : Boolean = True;

When this constant is set to False, BROWSER does not read the data record associated with each key to display in the browse screen. As a result, the **DatS** parameter passed to the user **BuildARow** routine is uninitialized and should not be referenced. Setting **<ReadDataRecord>** False improves performance when the key string alone provides adequate information to create the display string, or the information to be displayed will be obtained by indirect means.

RefreshFunc : Pointer = Nil;

The address of a user-defined routine that detects fileblock modification by other workstations. See "Automatic Screen Refresh" earlier in this section for more information.

RefreshPeriod : Word = 90;

Number of clock ticks that the **<RefreshPeriodically>** function waits before checking whether another workstation has modified the fileblock. There are approximately 18.2 clock ticks per second, so the default delay is about 5 seconds. If you reduce **<RefreshPeriod>** the browser will detect changes made by other workstations more quickly, but file and message activity on the server will increase proportionally.

RetriesOnLock : Integer = 50;

Number of retries in case of a lock error that occurs when the browser calls **<BTFindKeyAndRef>**, **<BTNextKey>**, **<BTPrevKey>**, **<BTReadLockFileBlock>**, or **<BTSearchKey>**.

RowsToJump : Integer = 0;

Determines how many rows the display scrolls when the highlight bar is at the bottom of the screen and the down arrow is pressed (or conversely, when the bar is at the top and up arrow is pressed). The default value, 0, means that the display will scroll by half of the window height. A common value for **<RowsToJump>** is 1, which will generate smooth scrolling, at the cost of performance, especially in multi-user environments.

```
UseReadLock : Boolean = False;
```

When <UseReadLock> is set to True, BROWSER places a readlock on the fileblock as it builds each page of the browse display. If it can't obtain the readlock in <RetriesOnLock> attempts, Browse exits and returns the function result 2. A readlock is useful in this situation to prevent another workstation from placing a fileblock lock during the display update, which would cause the browser to pass a <Ref> number of -1 to the BuildARow routine for one or perhaps all the records.

Types

```
BKtype = BKnone..BKuser9;
```

<BKtype> defines the class of actions performed by the browser. It includes such actions as cursor up and down, item select, and browser exit. See the constants above for further details.

```
RowRec =  
  record  
    IKS : IsamKeyStr;           {Record key}  
    Ref : LongInt;             {Record reference number}  
    Row : String[MaxCols];     {String to display based on record contents}  
  end;
```

BROWSER uses this type to manage the record display. The <Browse> function declares an array of <RowRec>, portions of which it initializes itself and other parts which are initialized by a routine the application supplies. In particular, <Browse> initializes the <IKS> and <Ref> fields. Then it calls a user-defined routine, pointed to by <ProcBuildARow>, which computes the <Row> string. Finally, <Browse> calls another user-defined routine, pointed to by <ProcDisplayARow>, which writes the <Row> string to the screen.

Variables

```
BrowseHelpIndex : Word;
```

Contains the topic number to be passed to the help routine when it is activated. If help is activated, the application should initialize this variable just prior to each call to <Browse>.

```
BrowseHelpPtr : Pointer;
```

Points to a routine that will be called when the help key is pressed. By default, this pointer is Nil and no action occurs.

```
BrowseKeyPtr : Pointer;
```

Points to a routine that reads each keystroke while in the browser. By default, this points either to <BrowseReadKey> or to ReadKeyWord in TPCRT or OPCRT, routines that return both a scan code and an ASCII character. The application can point this to another routine that returns a key and perhaps performs some background task while awaiting input.

Declaration

```
function AddBrowseCommand(Cmd : BKtype;  
                          NumKeys : Byte;  
                          Key1, Key2 : Word) : Boolean;
```

Purpose

Modify a keystroke assignment.

Description

You can make three kinds of changes to the key assignments. First, you can add new keystrokes for exit or special task commands. Second, you can disable a particular command. And third, you can change a key assignment. We'll give some examples of each.

<Cmd> is the browser command number to change. <NumKeys> is 1 or 2, specifying whether <Key1> and/or <Key2> contain keystroke values. The key parameters contain key codes consistent with those returned by <BrowseReadKey>. However, the high byte of each word (the scan code) is ignored unless the low byte (the ASCII character) is zero.

Let's say that you want to let a user browse through a database and press to specify that a particular record should be deleted. To implement this, you need to map the key to one of the exit commands:

```
Status := AddBrowseCommand(BKuser0, 1, $5300, 0);
```

The first parameter is the command code. <BKuser0> is the first available user-defined exit command, so we select that here. When the user presses , <Browse> will return the value <BKuser0> in its <ExitKey> parameter.

The second parameter specifies how many keys follow. Acceptable values here are limited to 1 or 2. You would only specify 2 if you wanted the user to press something like <CtrlQ><CtrlR>, two separate keystrokes for a WordStar-like command. In the example, is just one key so we specify 1. The scan code and ASCII character for follow in the next parameter; we looked up the value in the chart in Turbo Pascal's manual. The final parameter is ignored in this case, because <NumKeys> is 1. Another character code would go here if needed.

<AddBrowseCommand> may return False for three reasons: if the table used to hold the commands is full (it will hold up to 200 keystrokes for all the commands together); if the <NumKeys> value is anything but 1 or 2; or if the new keystroke sequence would lead to an ambiguous interpretation of an existing command. Once your program is checked out, you can reasonably expect <AddBrowseCommand> to return True.

The same technique would be used to enable a special task routine for <Browse>. In that case, you would choose a command code in the range <BKtask0>..<BKtask9>.

AddBrowseCommand

Now let's suppose that you wanted to disable a command. You might want to disable the <Up> and <Down> arrow keys and restrict the user to moving a page at a time. The following sequence would do that:

```
Status := AddBrowseCommand(BKnone, 1, $4800, 0); {Disable <Up>}
Status := AddBrowseCommand(BKnone, 1, $5000, 0); {Disable <Down>}
```

The command code <BKnone> is a special value that means "no action should occur when this key is pressed", i.e., the keystroke should be ignored. Again, there is one keystroke for each command and we've looked up the scan codes for the characters in Turbo Pascal's chart.

Finally, let's say that you want to change an existing key assignment. For example, maybe you want <CtrlPgDn> to do the same thing as <PgDn>, and <CtrlPgUp> to do the same thing as <PgUp>:

```
Status := AddBrowseCommand(BKpageDown, 1, $7600, 0);
Status := AddBrowseCommand(BKpageUp, 1, $8400, 0);
```

When <AddBrowseCommand> scans the key table, it will see that <CtrlPgDn> is already assigned to <BKlastRec> (move to last record) and simply change its assignment to <BKpageDown>. Similarly, <CtrlPgUp> will be assigned to <BKpageUp> rather than <BKfirstRec>. In both cases, the amount of space remaining in the table for expansion will be unchanged.

See Also

BrowseReadKey

Declaration

```
function Browse(IFBPtr : IsamFileBlockPtr;
  VarRec : Boolean;
  KeyNr : Integer;
  LowKey : IsamKeyStr;
  HighKey : IsamKeyStr;
  StartScreenRow : Integer;
  NrOfRows : Integer;
  var DatS;
  var DatLen : Word;
  var Ref : LongInt;
  var KeyStr : IsamKeyStr;
  var ExitKey : BKeyType;
  ProcSpecialTask : Pointer;
  ProcBuildaRow : Pointer;
  ProcDisplayaRow : Pointer) : Integer;
```

Purpose

Display fileblock records and allow user to scroll through them.

Description

<IFBPtr> points to an open fileblock. The fileblock may be opened for single user, network, and/or variable length modes. <VarRec> must be True if the fileblock contains variable length records, otherwise it must be False. <KeyNr> specifies which of the indexes to use for ordering the records displayed by the browser.

The browser will display all keys in the range from <LowKey> to <HighKey>, inclusive. Note how <HighKey> is interpreted. If, for example, <HighKey> is 'SMITH', then all of the following records would appear in the browse window: 'SMIT', 'SMITH', 'SMITHERS', 'SMITHSON'. Specifying 'SMITH'#0 for <HighKey> would prevent the last two from appearing.

The combination of <KeyStr> and <Ref> specifies the first record to be displayed and highlighted. If <KeyStr> is less than <LowKey> then the first record greater than or equal to <LowKey> will be displayed. If <KeyStr> is greater than <HighKey> then the largest acceptable key will be highlighted first.

<DatS> is a buffer at least as large as the largest (fixed or variable length) record to be displayed. <DatS> returns the selected record when the browser exits, but is also used as a temporary buffer while browsing. If variable length records are browsed, <DatLen> returns the length of the selected record.

<StartScreenRow> specifies the screen row where the first record will appear, and <NrOfRows> is the number of records visible at once. If <NrOfRows> is less than <MinRows>, it is increased to that value. Similarly, if <NrOfRows> exceeds <MaxRows>, it is reduced to that value.

<ProcBuildaRow> and <ProcDisplayaRow> are pointers to procedures used to create and display a row, respectively. These procedures are provided by the application, and they *must* have a procedure header that matches the example

Browse

declaration described later in the reference section. Furthermore, these procedures must be declared with the far compilation model, and they must not be nested inside of other procedures. The far model is activated with the {\$F+} compiler directive.

The procedure whose address is passed in <ProcSpecialTask> will be called when keys assigned to command codes <BKtask0> through <BKtask9> are pressed. If no special task is desired, pass Nil for this parameter. <ProcSpecialTask>, if used, must also point to a procedure whose header exactly matches the one described under <SpecialTask>, is global, and is compiled under the far model.

The command code that caused <Browse> to exit is returned in <ExitKey>. The first action that <Browse> is to take is also specified in this parameter. If no action is to occur, <ExitKey> must be initialized to <BKnone> (= 0).

See the constants defined above for descriptions of each action that is possible while within the browser. The user will remain within <Browse> until one of the exit keys is pressed, after which it will return to the calling program.

<Browse> may return any of three function results:

- 0 Success
- 1 No keys found in the range <LowKey>..<>HighKey>
- 2 FILER error of class 2 or higher (check <IsamError>)

When <Browse> returns the success code of zero, the value of <ExitKey> can be checked to determine what action to take next. If <ExitKey> <> <BKquit> (any exit key but Escape was pressed), then <DatS> contains the selected data record, <Ref> its record number, and <KeyStr> its key string. When <ExitKey> = <BKquit>, all of these parameters are left undefined.

<Browse> neither clears the screen on entry nor restores it on exit. These actions are the responsibility of the calling program.

Example

See the introduction to this section for a simple example of using <Browse>. Also see NETDEMO.PAS and SIMPDEMO.PAS for more thorough examples.

See Also

BrowseAgain
DisplayARow

BuildARow
SpecialTask

Declaration

```
function BrowseAgain(IFBPtr : IsamFileBlockPtr;
                    VarRec : Boolean;
                    KeyNr : Integer;
                    LowKey : IsamKeyStr;
                    HighKey : IsamKeyStr;
                    StartScreenRow,
                    NrOfRows : Integer;
                    var HighlightedRow : Integer;
                    var HorizOfs : Integer;
                    var DatS;
                    var DatLen : Word;
                    var Ref : LongInt;
                    var KeyStr : IsamKeyStr;
                    var ExitKey : BKType;
                    ProcSpecialTask : Pointer;
                    ProcBuildaRow : Pointer;
                    ProcDisplayaRow : Pointer) : Integer;
```

Purpose

Browse a fileblock, returning to same location as previous call.

Description

This routine works just like <Browse>, but it is often more suitable for applications that call the browsing routine repeatedly in a loop. <BrowseAgain> can retain the exact contents of the browse window, including the position of the highlight bar, from one call to the next. By contrast, <Browse> recomputes the relative position of the records within the browse window for each call, which can lead to disconcerting visual effects when the routine is called within a loop.

<BrowseAgain> takes two parameters in addition to those of <Browse>: <HighlightedRow>, which specifies the relative screen row where the highlighted bar should appear, and <HorizOfs>, which specifies the amount of horizontal scrolling. If you maintain the values of these parameters between successive calls to <BrowseAgain>, the browse display will remain the same when <BrowseAgain> is recalled, instead of scrolling to values that it computes internally.

Example

```
var
  PF : IsamFileBlockPtr;
  Pages : LongInt;
  BrowseStatus : Integer;
  P : PersonDef;
  Len : Word;
  RefNr : LongInt;
  KeyStr : IsamKeyStr;
  ExitCmd : BKType;
  CurrentRow : Integer;
  HorizOfs : Integer;
...
  CurrentRow := 1;
  HorizOfs := 0;
```

BrowseAgain

```
RefNr := 1;
KeyStr := '';
ExitCmd := BKNone;
repeat
  BrowseStatus :=
    BrowseAgain(PF, False, 1, '', #255, 2, 20,
      CurrentRow, HorizOfs,
      P, Len, RefNr, KeyStr,
      ExitCmd, Nil, @BuildARow, @DisplayARow);
  if BrowseStatus <> 0 then begin
    {Error handling}
  end else begin
    case ExitCmd of
      {Handle various exit commands}
    end;
  end;
until ExitCmd = BKquit;
```

Shows how to initialize the CurrentRow and HorizOfs variables prior to a repeat loop that calls <BrowseAgain>. The case statement will typically handle commands to add a new record, delete a record, search, and so on. See SIMPDemo and NETDemo for complete examples.

See Also

Browse

Declaration

```
function BrowseReadKey : Word;
```

Purpose

Waits for a key to be pressed and returns its key code.

Description

For normal keystrokes (the letter 'a', for example), <BrowseReadKey> returns the ordinal value of the ASCII character (\$61 for 'a'). For extended keystrokes, it returns the scan code of the key in the high byte, and zero in the low byte (e.g., \$4800 for <Up> arrow). These return values are consistent with the way that <Browse> expects to read keys.

This function is interfaced by BROWSER so that programmer-defined background tasks may call it for basic keyboard input. See "Keyboard Handling" above for more information.

BuildARow

Declaration (name may be chosen as desired)

```
procedure BuildARow(var RR : RowRec;  
                    KeyNr : Integer;  
                    var DatS;  
                    DatLen : Word);
```

Purpose

Model routine called by <Browse> to convert a record to a display string.

Description

The BROWSER unit does not interface a routine by this name; you must declare and implement a routine that matches the one shown here. The routine must be global and compiled with the far model.

On entry to <BuildARow>, the field <RR.IKS> has already been initialized with the key string of the current record, and the field <RR.Ref> contains the current record number. The parameter <DatS> contains the complete data record. If <RR.Ref> equals -1, <RR.IKS> and <DatS> are uninitialized because the corresponding record was locked. If the global typed constant <ReadDataRecord> is False, <DatS> and <DatLen> are not initialized.

Prior to exiting, the procedure must initialize the field <RR.Row>. This field is a string no longer than <MaxCols> that contains information to be displayed for the record. The string may be wider than the physical screen, as long as the <DisplayARow> routine is prepared to display a portion of the string and support horizontal scrolling.

Example

See the introduction to this section.

See Also

Browse

DisplayARow

DisableBrowseMouse / DisableFiltering

Declaration

```
$(IFDEF UseMouse)  
procedure DisableBrowseMouse;
```

Purpose

Disable mouse control of the browser.

Description

Call this routine to temporarily disable mouse control within the BROWSER unit. Note that this routine points <BrowseKeyPtr> to the ReadKeyWord routine from Turbo Professional or Object Professional and disables mouse event handling in TPMOUSE or OPMOUSE.

See Also

EnableBrowseMouse

Declaration

```
procedure DisableFiltering;
```

Purpose

Disable record filtering.

Description

The next call to <Browse> will display all records whose keys fall in the range from <LowKey> to <HighKey>.

See Also

EnableFiltering

DisplayARow

Declaration (name may be chosen as desired)

```
procedure DisplayARow(var RR : RowRec;  
                      KeyNr : Integer;  
                      RowNr : Integer;  
                      StartRow : Integer;  
                      HighLight : Boolean;  
                      var HorizOfs : Integer);
```

Purpose

Model routine called by <Browse> to display a row string.

Description

The BROWSER unit does not interface a routine by this name; you must declare and implement a routine that matches the one shown here. The routine must be global and compiled with the far model.

The field <RR.Row> has a string formatted by <BuildARow> that is to be displayed by <DisplayARow>. <StartRow> contains the first row of the browser display, and <RowNr> reflects the relative position of the row. The absolute row where the record should be written is then <StartRow>+<RowNr>-1.

When <Highlight> is True, the record is currently selected. This should be indicated by the use of reverse video or some other special color.

<HorizOfs> is used to control horizontal scrolling. If the entire display string fits within the screen, this parameter may be ignored. Otherwise, note that <Browse> increments this parameter for every press of the <Right> arrow key and decrements it for every press of <Left>. The <DisplayARow> routine may modify the value of <HorizOfs> to constrain horizontal scrolling to a desired range.

Example

See the introduction to this section. SIMPDemo and NETDemo show how to implement horizontal scrolling.

See Also

Browse

BuildARow

EnableBrowseMouse / EnableFiltering

Declaration

```
{IFDEF UseMouse}  
procedure EnableBrowseMouse;
```

Purpose

Enable mouse control of the browser.

Description

This routine points <BrowseKeyPtr> to the ReadKeyOrButton routine from Turbo Professional or Object Professional and enables mouse event handling in TPMOUSE or OPMOUSE. See "Mouse Support" earlier in this section for more information.

See Also

DisableBrowseMouse

Declaration

```
procedure EnableFiltering(ValidateFunc : Pointer);
```

Purpose

Enable record filtering.

Description

<ValidateFunc> is the address of a global, far function whose declaration matches the following prototype:

```
{F+}  
function ValidateARecord(IFBPtr : IsamFileBlockPtr;  
                        KeyNr : Integer;  
                        Ref : LongInt;  
                        var KeyStr : IsamKeyStr;  
                        NetUsed : Boolean) : Boolean;
```

Before accepting any record, the browser calls the validation function to determine whether it meets the filtering criteria. The validation function should return True to accept the record for display; False otherwise. Note that the record has not been read from the data file at the time the validation routine is called.

See "Record Filtering" earlier in this section for more information.

See Also

DisableFiltering

IsFilteringEnabled / SpecialTask

Declaration

function IsFilteringEnabled : Boolean;

Purpose

Returns True if a record validation function is enabled.

Description

This routine is primarily for status reporting.

See Also

EnableFiltering

Declaration (name may be chosen as desired)

```
procedure SpecialTask(IFBPtr : IsamFileBlockPtr; var DatS;  
    Ref : LongInt; IKS : IsamKeyStr;  
    KeyNr : Integer; var Command : BKType;  
    var ExitCode : Integer; DatLen : Word);
```

Purpose

Model routine called by <Browse> when a special task key is pressed.

Description

The BROWSER unit does not interface a routine by this name; you must declare and implement a routine that matches the one shown here. The routine must be global and compiled with the far model.

<IFBPtr>, <DatS>, <Ref>, <IKS>, <KeyNr>, and <DatLen> describe the current fileblock and record at the time the special task key was pressed. These may be used to further define the actions of the special task.

<Command> contains the command code that caused this procedure to be called. It may take on the values <BKtask0>..<BKtask9>. There is an important special meaning for this parameter as well. Its value upon return from <SpecialTask> determines what the browser will do next. For example, if the special task determines that the browser must execute a <PgDn> action after returning, <Command> should contain <BKpageDown> upon exit. If no action is required, be sure to set <Command> to <BKnone> (= 0) before exiting the special task routine. Even repeated calls to the special task or recursive calls to <Browse> are possible. If you make a recursive call to <Browse> remember that each call consumes about 4000 bytes of stack space.

<ExitCode> is used to let <Browse> know if an error occurred. Zero means no error. Any other value causes <Browse> to terminate and return this value as its function result.

D. REBUILD/VREBUILD: Rebuilding a Fileblock

Unfortunately, it's just too easy to corrupt a database, particularly one running on a network. If the server crashes, or the power goes out, or a program error causes the system to hang, information will probably remain buffered in memory. At this point the integrity of the database is highly suspect: was a new record written to disk, while its index entries remained buffered? Was the index partially updated, while some B-tree pages remained in memory?

B-Tree Filer, unlike some database managers, can at least detect this form of corruption. Whenever data is buffered, B-Tree Filer writes out a flag that remains set until the buffers are guaranteed to be flushed (usually when the fileblock is closed). If this flag is found to be set when a fileblock is opened, B-Tree Filer returns an error code of 10010, indicating that the fileblock must be "rebuild".

The <RebuildFileBlock> procedure was designed especially to make it easy for you to recover from this error. This routine reads the existing data file (ignoring the potentially damaged index file), builds a new data file using just the non-deleted records, and creates a complete new index file.

There are other situations where this same operation can be useful:

- many records have been deleted, and no significant number of records will be added in the near future. In this case, we call the rebuild a "purge", since it removes unused space from both the data and index files.
- the number or type of keys is to be changed. Since the rebuild generates all the indexes from scratch, it's a perfect opportunity to reindex the data.

Reconstruction of a fileblock is inherently a single-user activity. No workstation, including the current one, should have the fileblock open at the time when a reconstruction is to start. If the fileblock is held open by another user, <RebuildFileBlock> will fail immediately because it opens the data file in a non-shareable mode. If the fileblock has been opened by the current workstation, unpredictable problems may occur.

To use the rebuild routines in a program, just add REBUILD or VREBUILD to your uses statement, following the FILER unit. VREBUILD also depends on the VREC unit. Most of REBUILD's work is actually done by the REORG unit, and most of VREBUILD's work is done by the VREORG unit.

Note that a reconstruction is possible only if the first four bytes of each data record were reserved for B-Tree Filer's use, and were initialized to zero by the application

whenever it added a record. We've been harping on this issue throughout the manual--let's hope it doesn't catch you when the time has come for a rebuild.

As an aside, we should note that there are other methods sometimes used to denote deleted data records. One such method is to zero out the entire record (except its first four bytes) when the record is deleted. This is done with the sequence: <BTDeleteRec>, <BTGetRec>, zero the fields, <BTPutRec>. During a rebuild, the deleted records can be recognized by the fact that all fields are null. This approach has the additional advantage of improved data security: the deleted records are completely obliterated. B-Tree Filer's REBUILD unit does not support this approach; you could use the REORG unit described in Section 6.E to rebuild a fileblock that doesn't mark each record with an initial zero, but does clear out the rest of the record as described.

REBUILD and VREBUILD Identifiers

This section describes all identifiers exported by the REBUILD and VREBUILD units. The routines are so similar that they can be described in tandem.

Also see constant <AddNullKeys> and variable <IsamReXUserProcPtr> described in Chapter 5. Both of these identifiers affect the behavior of REBUILD and VREBUILD.

Declaration (name may be chosen as desired)

```
function BuildKey(var DatS; KeyNr : Integer) : IsamKeyStr;
```

Purpose

Model routine called by rebuild routines to generate a key string from the given record.

Description

The rebuild units do not interface a routine by this name; you must declare and implement a routine that matches the one shown here. The routine must be global and compiled with the far model.

Such a function is called for each index and each record in the reconstructed fileblock. Note that the routine is called with index number 1 for each record, then with index number 2 for each record, and so on. This sequence tends to take best advantage of the index page buffers, leading to improved rebuild performance.

In order to treat the untyped var parameter <DatS> as a record of your chosen type, you may use typecasting or an absolute variable. To implement the latter approach, simply declare a variable local to the <BuildKey> routine as follows:

```
MyDat : MyRecordType absolute DatS;
```

<BuildKey> must build the key string associated with the data record <DatS> for index number <KeyNr>. Be sure that the returned string does not exceed the allowed length for the index; if it does, the rebuild routine will generate an error and exit prematurely.

If <BuildKey> returns an empty string and the global typed constant <AddNullKeys> (from the FILER unit) is set to False, the rebuild routine will not add a key for that index and record.

Example

See the CreateKey function in Section 4.C.

RebuildFileBlock / RebuildVFileBlock

Declaration

```
procedure RebuildFileBlock(FBName : IsamFileBlockName;  
                           DatSLen : LongInt;  
                           NumberOfKeys : Integer;  
                           IID : IsamIndDescr;  
                           FuncBuildKey : Pointer);  
  
procedure RebuildVFileBlock(FBName : IsamFileBlockName;  
                            DatSLen : LongInt;  
                            NumberOfKeys : Integer;  
                            IID : IsamIndDescr;  
                            FuncBuildKey : Pointer);
```

Purpose

Rebuild a fileblock.

Description

This high-level procedure rebuilds a fileblock of name <FBName>. The fileblock should not be open on any workstation, including the current one, at the time of the call. <FBName> must contain either one, two, or three valid DOS filenames, separated by semicolons, with optional drive and pathname specifications. No extensions should be specified since these are automatically appended using <DatExtension> (default='DAT') for the data file, <IxExtension> (default='IX') for the index file, and <SavExtension> (default = 'SAV') for the save file. Note that the save file may be located on a different drive from the data file if disk capacity is tight. See the discussion of type <IsamFileBlockName> near the beginning of Chapter 5 for more information about how to specify <FBName>.

The rebuild routine executes the following steps:

1. Renames or copies the '.DAT' file to '.SAV' if no '.SAV' file already exists.
2. Calls <BTCreateFileBlock> with name <FBName>, record length <DatSLen>, and <NumberOfKeys> keys as described by <IID>. Opens the new fileblock in non-network mode using <BTOpenFileBlock>.
3. Reads every non-deleted record from the '.SAV' file and adds it to the new fileblock using <BTAddRec>.
4. For every index, reads each record from the new '.DAT' file and adds the key to the new fileblock by calling <BTAddKey>.
5. Closes the fileblock.
6. If successful, erases the '.SAV' file.

While completing step 3, the rebuild routine allocates two heap buffers each large enough to hold the largest record in the fileblock. Be sure to reserve adequate heap space.

RebuildFileBlock / RebuildVFileBlock

In order to carry out step 4, <RebuildFileBlock> calls a routine made known to it through the pointer parameter <FuncBuildKey>. (Remember that B-Tree Filer never has knowledge of the actual contents of any record.)

<FuncBuildKey> must point to a function with a header identical to the example routine described on the <BuildKey> page of this section. The function must be compiled under the far model (by using the {\$F+} compiler directive or by declaring it in the interface section of a unit), and must not be nested inside of any other procedure or function.

If a duplicate primary key is encountered during the rebuild, the corresponding data record and all keys entered for it to that point are deleted. The contents of such data records along with their keys are written to a text file with the '.MSG' extension. This file may be examined later with the intent of performing further manual reconstruction. If no duplicate primary keys are noted, no '.MSG' file is created.

The rebuild procedure is aborted immediately if a severe I/O error occurs during the reconstruction. You should not delete the '.SAV' file; another run with the rebuild routine must read it again.

The <IsamReXUserProcPtr> variable interfaced by the FILER unit may be used to display status during the rebuild, or to check the keyboard for a user-initiated abort of the operation.

Before calling either rebuild routine, be sure to call <BTInitIsam>. Before calling <RebuildVFileBlock>, be sure to call <CreateVariableRecBuffer> or <SetVariableRecBuffer>.

Example

```
uses Filer, Rebuild;
const Key1Len = 30; Key2Len = 5; Key3Len = 15;
type PersonDef = record ... end;
var Pages : LongInt; IID : IsamIndDescr;
{$F+}
function CreateKey(var P; KeyNr : Integer) : IsamKeyStr;
begin ... end;
begin
  Pages := BTInitIsam(NoNet, 40000, 0);
  IID[1].KeyL := Key1Len; IID[1].AllowDupK := False;
  IID[2].KeyL := Key2Len; IID[2].AllowDupK := True;
  IID[3].KeyL := Key3Len; IID[3].AllowDupK := True;
  RebuildFileBlock('TEST', SizeOf(PersonDef), 3, IID, @CreateKey);
  if not IsamOK then begin
    (Error handling)
  end;
  BTExitIsam;
end.
```

This is a complete program to reconstruct the fileblock for the introductory example described in Section 4.C. CreateKey is the same function given there.

E. REORG/VREORG: Reorganizing a Fileblock

Reorganizing is a process similar to rebuilding, but one that is used in different circumstances. Reorganization is used when the size of each data record or its format must be changed. The B-Tree Filer REORG unit also provides a way to import some types of foreign data into the fileblock format.

The procedures <ReorgFileBlock> and <ReorgVFileBlock>, contained in the REORG and VREORG units, implement these ideas for fixed and variable length fileblocks, respectively. <ReorgFileBlock> will read any file of fixed length records, whether or not it was created by B-Tree Filer. (It always skips over the first record, however.) <ReorgVFileBlock> is specific to the format used by B-Tree Filer's variable length records.

By taking advantage of user-supplied routines to perform the data validation and conversion of each record, the REORG and VREORG units are very flexible and should be useful in many situations.

REORG depends on only the FILER unit. VREORG depends on both the FILER and VREC units.

REORG and VREORG Identifiers

This section describes all identifiers exported by the REORG and VREORG units. The routines are so similar that they can be described in tandem.

ConvertRecord / ConvertVRecord

Declaration (name may be chosen as desired)

```
function ConvertRecord(var DatSold; var DatSNew; Len : Word) : Boolean;  
function ConvertVRecord(var DatSold; var DatSNew; var Len : Word) : Boolean;
```

Purpose

Model routine called to convert and verify a data record during reorganization.

Description

The reorganization units do not interface a routine by this name; you must declare and implement a routine that matches the one shown here. The routine must be global and compiled with the far model. Note that the required declaration varies depending on whether a fixed length record or variable length record fileblock is being reorganized.

<ConvertRecord> must return True if the record is to be added to the reorganized fileblock, or False if it is to be skipped. For a variable length record, <Len> specifies the length of the input record <DatSold>. If the converted record has a different length, <ConvertRecord> must return the new length in <Len>. Note that <Len> is also passed in for a fixed length record fileblock; this value of course cannot be changed for a fixed length record.

When <ReorgFileBlock> calls the <ConvertRecord> routine, it passes *all* records to the routine, even those which by B-Tree Filer's standards may represent deleted data. This is done to allow records from foreign formats to be imported into B-Tree Filer. The conversion routine should validate all records, and return False for those that should not be imported.

By contrast, <ReorgVFileBlock> calls the conversion routine only for records that are not considered to be deleted by B-Tree Filer's standards. <ReorgVFileBlock> may *not* be used to import foreign variable length record files such as comma-delimited text files.

Note that the <ConvertRecord> routine is never called for the first record in a fileblock, since that usually represents a system record.

Example

See the example for <FixToVarFileBlock> in the following section.

ReorgFileBlock / ReorgVFileBlock

Declaration

```
procedure ReorgFileBlock(FBName : IsamFileBlockName;
                        DatSlen : LongInt;
                        NumberOfKeys : Integer; IID : IsamIndDescr;
                        DatSlenOld : LongInt;
                        FuncBuildKey : Pointer; ProcChangeDatS : Pointer);

procedure ReorgVFileBlock(FBName : IsamFileBlockName;
                        DatSlen : LongInt;
                        NumberOfKeys : Integer; IID : IsamIndDescr;
                        DatSlenOld : LongInt;
                        MaxDiffBytes : Word;
                        FuncBuildKey : Pointer; ProcChangeDatS : Pointer);
```

Purpose

Reorganize a fileblock.

Description

This high-level procedure converts a fileblock of name <FBName> to a new format. The fileblock should not be open on any workstation, including the current one, at the time of the call. <FBName> must contain either one, two, or three valid DOS filenames, separated by semicolons, with optional drive and pathname specifications. No extensions should be specified since these are automatically appended using <DatExtension> (default='DAT') for the data file, <IxExtension> (default='IX') for the index file, and <SavExtension> (default = 'SAV') for the save file. Note that the save file may be located on a different drive from the data file if disk capacity is tight. See the discussion of type <IsamFileBlockName> near the beginning of Chapter 5 for more information about how to specify <FBName>.

<DatSlen> is the record length (section length for a variable length record fileblock) of the new fileblock. <DatSlenOld> is the record length (section length) for the original fileblock. For a variable length record fileblock, <MaxDiffBytes> is the maximum number of bytes by which the total record length of a reorganized record can differ from the original record. If the new records are smaller, pass zero for <MaxDiffBytes>.

The reorganization routine performs the following steps:

1. Renames the '.DAT' file to '.SAV' if no '.SAV' file already exists.
2. Calls <BTCreatFileBlock> with name <FBName>, record length <DatSlen>, and <NumberOfKeys> keys as described by <IID>, then opens the fileblock by calling <BTOpenFileBlock> in single user mode.
3. Reads every record from the '.SAV' file.
4. Calls the conversion routine, and if requested, adds the converted record to the new fileblock using <BTAddRec>.

5. For every index, reads every record from the new '.DAT' file and adds the key to the fileblock by calling <BTAddKey>.
6. Closes the fileblock.
7. If successful, erases the '.SAV' file.

While completing steps 3 and 4, the routine allocates two heap buffers each large enough to hold the largest record in the fileblock. Be sure to reserve adequate heap space.

To carry out step 4, <ReorgFileBlock> depends on a user-written function to build a new data record from the contents of the old one and return a value determining whether the data record should be added to the new fileblock. And for step 5, it depends on another user-written function to build the key for each record.

<FuncBuildKey> and <FuncChangeDatS> contain the addresses of these functions, which must be global and compiled under the far model. See <BuildKey> in Section 6.D for a description of the first routine. See <ConvertRecord> in this section for more information about the second one.

In contrast to <RebuildFileBlock> and <ReorgVFileBlock>, <ReorgFileBlock> passes *all* data records (even those not initiated by four null bytes) to the conversion routine. This is done to allow fixed length data files from other sources to be reorganized. The user-written conversion routine must determine the validity of each data record by whatever means necessary, and return True to accept the record, or False to skip it.

If a duplicate primary key is encountered during the reorganization, the corresponding data record and all keys entered for it are deleted. The contents of the data record along with its keys are written to a text file with the '.MSG' extension. This file may be examined later with the intent of performing further manual reconstruction. If no duplicate primary keys are noted, no '.MSG' file is created.

The procedure is aborted immediately if a severe I/O error occurs during the reorganization. The remaining '.SAV' file is not deleted, since it will be needed for another attempt.

The <IsamReXUserProcPtr> variable interfaced by the FILER unit may be used to display status during the reorganization, or to check the keyboard for a user-initiated abort of the operation.

Before calling either reorganization routine, be sure to call <BTInitIsam>. Before calling <ReorgVFileBlock>, be sure to call <CreateVariableRecBuffer> or <SetVariableRecBuffer>.

F. FIXTOVAR: Converting to Variable Length Records

This small unit has a single purpose: to convert an existing B-Tree Filer fileblock based on fixed length records to a new fileblock using variable length records. This action is especially appropriate if you want to add a new variable length field to an existing fileblock.

FIXTOVAR actually uses the REORG and VREORG units to do most of its work, so the documentation for those units is also applicable here. FIXTOVAR interfaces a single high level routine, described on the following page.

Declaration

```
procedure FixToVarFileBlock(FBName : IsamFileBlockName;  
    DatSlenFix : LongInt; DatSlenVar : LongInt;  
    NumberOfKeys : Integer; IID : IsamIndDescr;  
    FuncBuildKey : Pointer);
```

Purpose

Converts a fileblock based on fixed length records to variable length records.

Description

Internally, this procedure first calls <ReorgFileBlock> to expand and initialize each fixed length record so that the fileblock looks like a variable length fileblock, all of whose records are composed of a single section. Then it calls <ReorgVFileBlock> to "resection" each record into one or more smaller pieces.

<FBName> is the name of the existing fileblock. The name may contain up to three semicolon separated DOS filenames without extensions. (See the description of type <IsamFileBlockName> in chapter 5.) The first filename specifies the location of the existing data file and the destination for the new variable length data file. The second filename specifies the destination for the new index file (the old index file is not used). The third filename specifies the location for the "SAV" file, which is a backup copy of the old fixed-length data file. When the data file is large it may be necessary to use a different disk drive for storage of the backup data. If the data file is not found, the routine will use the SAV file immediately. If the data file is found, it is renamed or copied onto the SAV file before proceeding.

No fileblock of name <FBName> may be open at the time of the call to <FixToVarFileBlock>.

<DatSlenFix> is the length of the existing fixed length records. <DatSlenVar> is the desired section length for the variable length records. See Section 6.B for a discussion about how to pick <DatSlenVar>.

<NumberOfKeys> and <IID> specify the properties of the new index file. Note that it's possible to change the way the fileblock is indexed at the same time as the data file is reorganized. See the descriptions of type <IsamIndDescr> and procedure <BTCreateFileBlock> in Chapter 5 for more information.

<FuncBuildKey> is the address of a user-defined function that will return the key string for each record and each index as the index is rebuilt. See Section 6.D for more information.

<FixToVarFileBlock> may take advantage of any of the customization hooks described in Sections 6.D and 6.E. It can return any of the <IsamError> codes described there also.

FixToVarFileBlock

Example

Suppose that you have a fileblock containing fixed length records of size 200 bytes. Suppose also that you plan to add a variable length memo field to the end of these records, and that the memo field might contain anywhere from 1 character (a terminator) to 1000 characters. Following the guidelines given in Section 6.B, a suitable section length for the resulting variable length fileblock is 256 bytes.

The first step of the conversion is to make a variable length fileblock that doesn't contain the new memo field. You'd use `<FixToVarFileBlock>` to perform this step:

```
FixToVarFileBlock(FBName, 200, 256, 0, IID, Nil);
```

Note that we specify zero keys, since there's no need to generate the index twice. The IID variable can be uninitialized at this time since it won't be used. Similarly, the `<FuncBuildKey>` pointer can be Nil because it's never called.

The next step is to add the memo field, which we'll assume is initially empty for all the records. We'll also assume that a memo field is an array of characters terminated by `^Z`. Hence, an empty memo field is indicated by a single `^Z` character. To perform the next step, we use `<ReorgVFileBlock>`:

```
ReorgVFileBlock(FBName, 256, NumberOfKeys, IID, 256, 8,
  @BuildKey, @ChangeDat);
```

We aren't changing the section length, so `<DatSLen>` equals `<DatSLenOld>` equals 256. This time we want to generate the index file so we pass the desired number of keys, a properly initialized IID, and the address of an appropriate BuildKey routine. `<MaxDiffBytes>` is 8, a small value that allows for the addition of the empty memo field (`<MaxDiffBytes>` could theoretically be as small as 1 here). The `ChangeDat` function would be written as follows:

```
($F+)
function ChangeDat(var DatSold; var DatsNew; var Len : Word) : Boolean;
begin
  move(DatSold, DatsNew, Len); {transfer existing contents}
  NewRecType(DatsNew).MemoField[0] := ^Z; {initialize empty memo field}
  inc(Len); {indicate that new record is one byte bigger than old}
  ChangeDat := True; {all records are transferred}
end;
```

Here we've assumed that the new record type is declared as follows:

```
type
  NewRecType =
    record
      ... all the old fields ...
      MemoField : array[0..1000] of char;
    end;
```

Although each record grows by one byte during the second step of the conversion, each one still fits within a single 256 byte section. Records require more than one section only if we add more than about 50 characters of text to the memo field.

G. NUMKEYS: Numeric Keys

B-Tree Filer requires that all index keys take the form of a string. This offers the greatest flexibility in designing indexes: keys can be composed from portions of several record fields, they can be stored in compressed format, they can be logically inverted, and so on. Nevertheless, numeric values are often desirable for keys because they encode unique information in a compact format.

The NUMKEYS unit offers routines to convert numeric values to sortable key strings and back again. The strings produced are compact, and the exact numeric value may be recovered by calling the inverse routine. Although the encoded strings cannot be viewed directly, they are carefully designed to produce the correct sorting order when used in B-tree or other data files.

The NUMKEYS unit also contains several routines that can create "packed" key strings. Given a regular string of characters, these routines will return a compressed string, one in which each character is represented in either 4, 5 or 6 bits, rather than 8. In cases where the packed key strings can be used, you can thus obtain a savings of roughly 25-50% (e.g., a 50-character string can be represented in 25, 32, or 38 characters). As with the numeric key strings, these packed key strings cannot be viewed directly, but they will produce the correct sorting order when used in a B-tree. Complementary routines are also provided to unpack a packed key string.

NUMKEYS also contains one routine to invert the value of a key string. Such a key will sort in descending order. Inverting it a second time returns the original string.

Note that NUMKEYS does not support the AsciiZeroKeys define of the FILER unit. NUMKEYS is likely to generate key strings that contain embedded null characters, which cannot be reliably converted to ASCIIZ strings.

Special thanks to Scott Bussinger for the ideas behind some of the routines in this unit.

NUMKEYS Identifiers

This section describes all identifiers exported by the NUMKEYS unit. Because of the large degree of similarity between many of the routines, we'll describe them in groups rather than individually.

Types

```
String1 = String[1];  
String2 = String[2];  
String4 = String[4];  
String5 = String[5];  
String6 = String[6];  
String7 = String[7];  
String8 = String[8];  
String9 = String[9];  
String10 = String[10];
```

String types associated with the various numeric formats. Variables of type ShortInt are converted to type <String1>; Integer and Word are converted to type <String2>; LongInts to <String4>; Reals to <String6>; and Extendeds to <String10>. The intermediate string lengths (<String7>..<<String9>) are provided to allow shorter keys when the accuracy of floating point variables can be reduced.

Declaration

```
function DescendingKey(S : String; MaxLen : Byte) : String;
```

Purpose

Logically invert a string so that it will sort in descending order.

Description

Each character in the string <S> is inverted (using the assembly language NOT instruction) and the resulting string is padded to length <MaxLen> with \$FF bytes. This transformation causes the returned string to sort in exactly the reverse order of the original string.

Repeating the operation returns the original string, except that the final length will always be <MaxLen> and any padded bytes will contain the null character. To avoid this effect, it's always best to pad the original string with blanks to the maximum desired length.

The <InvertString> function in the ISAMTOOL unit (Section 6.H) works the same as <DescendingKey> except that two applications of <InvertString> always return the original string without padding.

Example

See the example for <InvertString>.

Numbers to Keys

Declaration

```
function ShortToKey(S : ShortInt) : String1;  
  {-Convert a shortint to a string}  
  
function IntToKey(I : Integer) : String2;  
  {-Convert an integer to a string}  
  
function WordToKey(W : Word) : String2;  
  {-Convert a word to a string}  
  
function LongToKey(L : LongInt) : String4;  
  {-Convert a longint to a string}  
  
function RealToKey(R : Real) : String6;  
  {-Convert a real to a string}  
  
function ExtToKey(E : Extended) : String10;  
  {-Convert an extended to a string}  
  
function BcdToKey(var B) : String10;  
  {-Convert a BCD real to a string}
```

Purpose

Convert numeric types to key strings.

Description

Each of these routines converts a numeric variable of the specified type to a string. The string is intended to be used as a key for a B-Tree Filer fileblock; it is not meant to be displayed, as the individual characters have values spanning the complete range from 0 to 255. String comparison operators (" $<$ ", " $>$ ", etc.) will return the expected results when comparing two converted strings.

As returned, each string has a length equal to the maximum allowed for its type. For example, `<RealToKey>` always returns strings that are 6 characters long. For numeric variables of type `ShortInt`, `Integer`, `Word`, `LongInt`, and `Real`, the entire string must be stored in order to preserve the meaning of the number.

`<ExtToKey>` is provided only if the `NUMKEYS` unit is compiled with the `{ $\$N+$ }` (numeric coprocessor) option enabled. This one routine may be used to convert IEEE floating point numbers of type `Single`, `Double`, `Extended`, and `Comp` to strings. Not all characters of the returned string are significant for the less precise types. You may use any of the following string types to hold the results of `<ExtToKey>` without losing any precision:

<code>Single</code>	: <code>String5</code> (min) - <code>String10</code> (max)
<code>Double</code>	: <code>String9</code> (min) - <code>String10</code> (max)
<code>Extended</code>	: <code>String10</code> (min/max)
<code>Comp</code>	: <code>String10</code> (min/max)

Slightly shorter strings (one less than the recommended minimum) may be used for `Singles`, `Doubles`, and `Extendeds` if you are willing to sacrifice some precision, however. We strongly recommend that you always use a `String10` for `Comps`.

The IEEE floating point standard embodied in the 8087 chip defines several special classes of numbers that aren't normally part of formal mathematics, but that are useful in the context of numerical analysis. Examples of such numbers are NAN (Not A Number), INF (positive infinity), and -0 (zero with a negative flavor).

When the floating point processor (or emulator) deals with these special numbers, it applies special rules. For example, it returns True when testing the assertion $(0 = -0)$, and the result of adding NAN to any normal number is NAN.

When such numbers are converted to strings, the special meaning is lost. NUMKEYS converts 0 to a different string than it does -0. Hence, although a numeric comparison of 0 and -0 returns equality, a string comparison does not. The string comparison does return the arguably logical result that -0 is less than 0.

These considerations should not pose a concern in normal situations. The special meanings are restored by the reverse conversion routines.

The parameter passed to <BcdToKey> must be a variable of type BCD as defined by the TPBCD and OPBCD units from Turbo Professional and Object Professional, respectively.

Example

Consider the PersonDef record described in Section 4.C and assume that we'd like to have an index for the Age field of the record, which is of type Integer. This index would have a maximum key length of 2 and would allow duplicates. To add a key for this index, we'd make the following call:

```
BTAddKey(PF, KeyNr, RefNr, IntToKey(P.Age));
```

To find a key based on age, we'd also need to convert an integer variable containing the age to search for into a string by calling <IntToKey>.

Keys to Numbers

Declaration

```
function KeyToShort(S : String1) : ShortInt;  
  (-Convert a string to a shortint)  
  
function KeyToInt(S : String2) : Integer;  
  (-Convert a string to an integer)  
  
function KeyToWord(S : String2) : Word;  
  (-Convert a string to a word)  
  
function KeyToLong(S : String4) : LongInt;  
  (-Convert a string to a longint)  
  
function KeyToReal(S : String6) : Real;  
  (-Convert a string to a real)  
  
function KeyToExt(S : String10) : Extended;  
  (-Convert a string to an extended)  
  
procedure KeyToBcd(S : String10; var B);  
  (-Convert a string to a BCD real)
```

Purpose

Convert a string back to a numeric type.

Description

These routines undo the original conversion to a key string. As long as the full length of the converted string was stored, the numeric value will be restored exactly.

<KeyToExt> is provided only if the NUMKEYS unit is compiled with the {\$N+} (numeric coprocessor) option enabled.

A variable of type BCD must be passed in the untyped parameter of <KeyToBcd>.

Declaration

```
function Pack4BitKey(Src : String; Len : Byte) : String;  
  (-Pack Src into sequences of 4 bits)  
  
function Pack5BitKeyUC(Src : String; Len : Byte) : String;  
  (-Pack Src into sequences of 5 bits. Alphas converted to upper case.)  
  
function Pack6BitKeyUC(Src : String; Len : Byte) : String;  
  (-Pack Src into sequences of 6 bits. Alphas converted to upper case.)  
  
function Pack6BitKey(Src : String; Len : Byte) : String;  
  (-Pack Src into sequences of 6 bits)
```

Purpose

Pack a string into less space.

Description

These routines all return a packed key string in which each character in the original string (<Src>) is represented in 4, 5 or 6 bits rather than 8. The packed key string is therefore roughly 25-50% smaller than the original.

<Len> is the length of the packed string to be returned. The proper value for this parameter will depend on (1) the maximum length of the unpacked string passed as the <Src> parameter and (2) the number of bits used for packing. The formulas used to calculate what <Len> should be are:

```
4-bits: (MaxLength) div 2 + Ord(MaxLength mod 2 <> 0)  
5-bits: (MaxLength*5) div 8 + Ord((MaxLength*5) mod 8 <> 0)  
6-bits: (MaxLength*6) div 8 + Ord((MaxLength*6) mod 8 <> 0)
```

For example, if you are using <Pack5BitKeyUC> (which uses 5 bits) and the maximum length of an unpacked key string is 50, then <Len> may be calculated as follows:

```
Len = (50*5) div 8 + Ord((50*5) mod 8 <> 0)  
    = 250 div 8 + Ord(250 mod 8 <> 0)  
    = 31 + Ord(2 <> 0)  
    = 31 + 1  
    = 32
```

Although this computation seems overly complicated, keep in mind that Turbo Pascal versions 5.0 and later, which support calculated constants, can make the necessary calculations for you at compile time. For example:

```
const  
  UnpackedLen = 50;  
  PackedLen = (UnpackedLen*5) div 8 + Ord((UnpackedLen*5) mod 8 <> 0);
```

Each of these routines makes important assumptions about what kinds of characters are in <Src>. You must therefore be certain that you choose the routine which is most appropriate for a given situation. The table below shows the set of characters that each routine can represent in the bits allotted it. In each case the numbers on the right indicate the actual values used to represent each character.

Pack Key Strings

Pack4BitKey

'(','),','+', '-','.' 1-5
'0'..'9' 9-15
all others 0

Pack6BitKeyUC

'!'..' ' 1-63
'A'..'Z' 33-58
'a'..'z' 33-58
all others 0

Pack5BitKeyUC

'A'..'Z' 1-26
'a'..'z' 1-26
all others 0

Pack6BitKey

'0'..'9' 1-10
'A'..'Z' 11-36
'a'..'z' 37-62
all others 0

As you can see, each packing scheme has distinct advantages and disadvantages:

<Pack4BitKey> is best for strings that contain only numbers plus the common punctuation marks associated with numbers.

<Pack5BitKeyUC> can represent only upper case alphabetic characters. (Lower case alphabets are automatically converted to upper case.)

<Pack6BitKeyUC> is similar to <Pack5BitKeyUC>, but it can represent numeric characters ('0'..'9') and punctuation marks (except "'") as well as upper case alphabetic characters. (Lower case alphabets are automatically converted to upper case.) Although it can represent more characters than <Pack5BitKeyUC>, <Pack6BitKeyUC> provides a lower degree of compression (roughly 25%).

<Pack6BitKey> gives the same degree of compression as <Pack6BitKeyUC>, but it can represent only numeric characters ('0'..'9') and alphabets, not punctuation marks. Its chief advantage over the other two routines is that it can distinguish between upper and lower case alphabetic characters. In situations where case sensitivity is important, it is thus the only one of the three routines that can be used.

Declaration

```
function Unpack4BitKey(Src : String) : String;  
  {-Unpack a key created by Pack4BitKey}  
  
function Unpack5BitKeyUC(Src : String) : String;  
  {-Unpack a key created by Pack5BitKeyUC}  
  
function Unpack6BitKeyUC(Src : String) : String;  
  {-Unpack a key created by Pack6BitKeyUC}  
  
function Unpack6BitKey(Src : String) : String;  
  {-Unpack a key created by Pack6BitKey}
```

Purpose

Unpack key strings.

Description

These routines unpack key strings created by <Pack4BitKey>, <Pack5BitKeyUC>, <Pack6BitKeyUC>, and <Pack6BitKey> respectively.

Note that the strings returned will not necessarily be identical to those originally passed to the packing routines. Any characters that the packing routines could not store in the allotted bits will be returned here as spaces, and lower case alphabetics ('a'..'z') will be returned as upper case characters unless <Pack6BitKey> was used.

The length of the unpacked string may also be greater than that of the original (if the original was shorter than the maximum length used to calculate the Len parameter passed to the packing routine); if so, the extra characters will all be blanks.

H. ISAMTOOL: Miscellaneous Routines

This is a collection of miscellaneous routines to extend the FILER unit, which for various reasons were felt to be unsuitable for inclusion in the FILER unit itself. Here's a brief overview of the routines interfaced by ISAMTOOL:

ExtendHandles

Increases the number of file handles available to an application

IsamErrorMessage

Returns a text string describing the specified error code

WSNrLogIn

Assigns a unique workstation number in network-independent fashion

WSNrLogOut

Returns a workstation number assigned by <WSNrLogIn> to the free pool

InvertString

Return a string that will sort in descending order

<IsamErrorMessage> can return strings in either English or German. By default it returns English strings. To enable German strings, edit ISAMTOOL.PAS and activate the GermanMessage conditional define. You can enable both EnglishMessage and GermanMessage if you like. Also change the typed constant <UsedErrorMessage> (which defaults to <English>) to <German> to complete the language switchover. If you enable both languages you can switch between them at runtime.

Declaration

```
procedure ExtendHandles(NumHandles : Word);
```

Purpose

Increase the number of usable file handles.

Description

<ExtendHandles> depends on a DOS function available for the first time with DOS 3.3. It will exit with <IsamError> set to 10190 if the DOS version is older.

<NumHandles> contains the desired number of file handles. If <NumHandles> is less than 21, <ExtendHandles> exits without doing anything (since DOS provides 20 handles by default). The maximum number of handles is 255, but a bug in DOS 3.3 limits the actual maximum to 254. (The <NumHandles> parameter is of type Word for compatibility with a Microsoft Windows 3.0 function for extending the number of handles.)

The CONFIG.SYS file on the system where the application runs must have a FILES= entry that is at least as large as the value of <NumHandles>. Again because of a bug in DOS 3.3, the FILES= entry should generally be set to at least <NumHandles>+1.

For workstations running under Novell NetWare, there is another configuration parameter that limits the number of available file handles to a default of 40. For recent versions of NetWare (2.1 or later) the FILE HANDLES= setting in SHELL.CFG specifies the maximum number of open files for a given workstation. See your NetWare Supervisor Reference for additional information. For earlier versions of NetWare the network shell must be patched to allow more open handles. Contact Novell for further information about the patch.

A new Handle table is allocated when <ExtendHandles> is called. The amount of memory required is <NumHandles>+47 bytes. In order to obtain this memory, the routine either uses existing free DOS memory or shrinks the Turbo Pascal heap to make some DOS memory available. This memory is automatically deallocated by DOS when the program ends.

<ExtendHandles> transfers any open files into the new handle table. Nevertheless, this routine should be called at the beginning of the program, preferably before any files are opened.

Note that DOS reserves the first five handles for standard devices. Hence, when using the standard 20 entry handle table, only 15 handles are available for other uses. B-Tree Filer needs two file handles for every single user fileblock opened in normal mode. In save mode, or when opened for network access, a fileblock requires three file handles.

ExtendHandles

<ExtendHandles> uses a documented DOS call supported only in versions 3.3 and later. For applications that must run under earlier versions of DOS, we recommend the public domain unit EXTEND, available on many bulletin boards and on the BPROGA forum of CompuServe. EXTEND works for DOS versions 2.0 and later by using different techniques depending on the DOS version.

Example

Assume that you wish to have 10 network fileblocks open at the same time, and still have 3 handles available for other uses. The number of required handles is therefore $10 \times 3 + 3 + 5 = 38$. CONFIG.SYS should contain the following entry:

FILES=39

After a successful call to the following code, the program will be able to open 38 or 39 handles:

```
ExtendHandles(38);  
if not IsamOK then begin  
  (Error handling)  
end;
```


Declaration

```
function InvertString(DS : String; MaxLen : Byte) : String;
```

Purpose

Inverts the string <DS> to generate a descending key string.

Description

This function has two specific properties that are useful in manipulating strings for fileblock indexes:

1. Given two strings A and B such that $A < B$, `<InvertString>` returns strings such that `InvertString(A) > InvertString(B)`.
2. Applying `<InvertString>` twice returns the original string, i.e., `InvertString(InvertString(A)) = A`.

By taking advantage of these two properties one can easily create fileblock indexes that are stored in descending order and also allow the original ASCII strings to be recovered.

<MaxLen> specifies the maximum length of a string in the index, typically the value passed in the <IsamIndDescr> variable when the fileblock was created. If the actual length of <DS> is larger than <MaxLen>, <InvertString> always returns the empty string.

The strings returned by one call to <InvertString> are not intended for display. Call <InvertString> a second time before viewing a key string.

Example

```
BTAddRec(PF, RefNr, PersonRec);
{Add a key to index 1, which is stored in descending order}
DKey := BuildKey(PersonRec, 1);
BTAddKey(PF, 1, RefNr, InvertString(DKey, MaxKeyLen));
...
Write('Enter key to search for: ');
ReadLn(DKey);
DKey := InvertString(DKey, MaxKeyLen);
BTSearchKey(PF, 1, RefNr, DKey);
if IsamOK then
    Writeln('The closest key found is ', InvertString(DKey, MaxKeyLen));
```

This example sketches the creation of a descending index for a fileblock. The normal ASCII key (a last name, for example) returned by the user-defined `BuildKey` routine is inverted before the key is added to the index file with `<BTAddKey>`. We assume here that index 1 of fileblock PF is allowed to store key strings up to <MaxKeyLen> characters long. Before searching for a key string, we also invert the key string. When a match is found, we invert the resulting string before displaying it.

IsamErrorMessage

Declaration

```
function IsamErrorMessage(ErrorNr : Integer) : String;
```

Purpose

Returns a message for the given error code.

Description

The value passed in <ErrorNr> is typically the value in the global variable <IsamError> after detecting that <IsamOK> is False.

<IsamErrorMessage> does not incorporate the file name of the associated fileblock. It is the application's responsibility to do so (see function <BTDataFileName> in the FILER unit). The messages returned by <IsamErrorMessage> generally match those found in Appendix A.

Be warned that calling <IsamErrorMessage> links about 2000 bytes of strings and code into your application.

Please note the comments in the introduction to this section regarding support for different languages.

Example

```
S := IsamErrorMessage(10190);
```

assigns the following to the string S: 'Extend handle function requires DOS 3.3 or later'.

Declaration

```
procedure WSNrLogIn(WSLogFileName : IsamFileBlockName);
```

Purpose

Generate a unique workstation number.

Description

This function can be called when the network operating system provides no automatic method to determine a unique workstation number. It should be called prior to calling <BTInitIsam>. See Section 2.F for more background.

<WSLogFileName> must contain a complete pathname to a file that is accessible to all workstations. <WSNrLogIn> attempts to open this file; if it isn't found, it will be created. The workstation number, which is in the range of 1 to <MaxNrOfWorkstations>, is assigned to the FILER unit's global variable <IsamWSNr>. The log file consumes <MaxNrOfWorkstations> bytes. The file is closed before <WSNrLogIn> returns.

<WSNrLogIn> opens the log file using the DOS mode "read/write, deny all", which effectively puts a physical lock on the entire file. (Note that this open mode requires DOS version 3.0 or later, and SHARE or an equivalent operating system service must be loaded.) <WSNrLogIn> automatically tries to open the file 10 times in case another station is also obtaining a number at the same time.

In order for the <WSNrLogIn> mechanism to work correctly, it is essential to call <WSNrLogOut> before the program ends. In fact, ISAMTOOL installs an exit procedure so that <WSNrLogOut> will be called even if the program terminates abnormally.

<WSNrLogIn> will generate <IsamError> 10305 if it is unable to access the log file (after retries), or error 10303 if the log table is full. It may also generate errors in the range 9500..9899, which correspond to Turbo Pascal IOResult codes after 9500 is subtracted.

Example

Assume that B-Tree Filer is to be initialized for <MsNet>, a network that has no automatic means to generate a workstation number. Using the following code, a workstation number is generated with the help of <WSNrLogIn>.

```
WSNrLogIn('N:\DATA\WSNUMBS.LOG');  
if not IsamOK then begin  
  {Error handler}  
end;  
Pages := BTInitIsam(MSNet, ...);
```

<WSNrLogIn> must be called before initializing B-Tree Filer so that the variable <IsamWSNr> contains a valid number.

WSNrLogOut

Declaration

procedure WSNrLogOut;

Purpose

Free a workstation number obtained via <WSNrLogIn> from the log file.

Description

This routine must be called before leaving a network application, sometime after calling <BTExitIsam>. <WSNrLogOut> returns the workstation number <IsamWSNr> to the free pool. The logfile is not deleted even if <IsamWSNr> is the last logged station. A network administrator can delete the log file when it is no longer needed after assuring that no workstations remain logged in.

ISAMTOOL's exit procedure automatically calls <WSNrLogOut> if the application halts abnormally. However, if the workstation crashes before exit code can be executed, the workstation number will remain assigned in the log file. If this happens enough times, all workstation numbers will be reserved and <WSNrLogIn> will fail with error 10303. To recover from a corrupted log file, simply delete the file when the application is not active. It may be a good idea to erase the log file whenever the network server is booted.

Example

```
BTExitIsam;
if not IsamOK then begin
  {Error Handler}
end;
WSNrLogOut;
if not IsamOK then begin
  {Error handler}
end;
```

The workstation number is freed from the logfile after the successful shutdown of B-Tree Filer.

I. MSORT: Sorting

Although sorting is a common database application, the supplied sort unit is not an integral part of the FILER unit in the same way that VREC or the other utility units are. The sort unit can be used to sort any kind of data, including B-Tree Filer data files. Some care is necessary when sorting data files, however. See the section "Sorting B-Tree Filer fileblocks" below for more details.

The virtual sort unit, MSORT, is an implementation of the "merge sort" algorithm. A merge sort allows you to sort more items than will fit in RAM at once, by sorting manageable portions of the input first, then merging these pre-sorted lists to form the final output. In theory, MSORT can sort two billion records at once. In practice, the number of records is limited by available disk or expanded memory (EMS) capacity, and the time you're willing to wait.

MSORT is used in a manner similar to the SORT unit of the Borland Database Toolbox and the TPSORT unit of our Turbo Professional package. If you've used either of these units before, you'll be quick to pick up MSORT.

MSORT exports several routines: <AutoSort>, <DoSort>, <PutElement>, <GetElement>, and <AutoSortInfo>. <AutoSort> is a high level sorting routine that automatically calculates various parameters needed by the sort system. <DoSort> is a lower level routine that gives you full control over all sort parameters. In most cases, you will use <AutoSort>, since it calculates these parameters and calls <DoSort>. <AutoSort> requires several parameters, as described in detail in the following sections. Two of the parameters tell it about the elements to be sorted--their size and the number to expect. One parameter specifies a pathname for any temporary files created by the sort process. Three parameters provide addresses of routines that MSORT will call during the course of execution--one routine that will load elements into MSORT's data structures, another that MSORT will call to compare pairs of elements, and a final one to return the sorted elements to your program.

The routines you provide to load and retrieve elements will call two of the other functions exported by MSORT. For each element to be sorted, your load routine will call <PutElement> to enter the data. For each sorted element retrieved, your routine will call <GetElement>.

The following small example shows how to organize an application that uses MSORT.

```

program Example;
uses
    Msort;
var
    Status : MSortStatus;
    NumToDo : Word;

{Routines called by MSORT must be both far and global}
{$F+}
procedure GetElem;
    {-Pass every element to be sorted into the Msort unit}
var
    W, I : Word;
begin
    for I := 1 to NumToDo do begin
        W := Random(MaxInt);
        if not PutElement(W) then begin
            Writeln('PutElement Error. ');
            Halt;
        end;
    end;
end;

procedure PutElem;
    {-Return each sorted element from the Msort unit}
var
    W : Word;
begin
    while GetElement(W) do
        Writeln(W);
end;

function Less(var X,Y) : Boolean;
    {-Compare elements being sorted}
begin
    Less := Integer(X) < Integer(Y);
end;
{$F-}

begin
    Write('Enter number of Integers to sort: ');
    ReadLn(NumToDo);
    Status := AutoSort(NumToDo, SizeOf(Integer), '', @GetElem, @Less, @PutElem);
    case Status of
        MSortSuccess      : Writeln('Success');
        MSortOutOfMemory  : Writeln('Insufficient memory');
        MSortDiskError    : Writeln('Disk Error: ', MSortIOResult);
        MSortOutOfDisk    : Writeln('Insufficient disk space for merge');
        MSortEMSError     : Writeln('EMS error');
    else
        Writeln('Unknown error: ', Ord(Status));
    end;
end.

```

Note that the GetElem, PutElem, and Less routines must be declared both far and global. Not observing this requirement will lead to an invalid sort sequence or even a program crash.

Sorting B-Tree Filer Fileblocks

In some cases, you may want to arrange the data records of a B-Tree Filer fileblock in physically sorted order. Typically you wouldn't want to arrange the records in the same order that an index already provides. This wouldn't provide an immediate advantage, and may actually slow down the process of adding keys.

If you do decide to sort a fileblock, you must consider some additional issues. The first issue is this: each deleted record within the data file contains a special tag used to manage a linked list of free space within the data file; sorting the data file corrupts this linked list. Of course, sorting the data file also invalidates the index file, which refers directly to the record numbers. And finally, the first record in the data file contains internal system information; it must remain first even after the sort. Hence, when sorting a B-Tree Filer fileblock, you must take special precautions, as shown in the following example.

```
type
  MyRecType =
    record
      Dele : LongInt;           {Deletion tag}
      {Other fields}
    end;
const
  DataName = 'MYDATA';        {Name of fileblock to sort}
  NumberOfKeys = 2;           {Number of keys in fileblock}

{$F+}
procedure GetElem;
  {-Pass every element to be sorted into the Msort unit}
var
  RecNum : LongInt;
  IFBPtr : IsamFileBlockPtr;
  Rec : MyRecType;
begin
  BTOpenFileBlock(IFBPtr, DataName,
    False, False, False, False); {Open only if closed}
  for RecNum := 1 to BTFilen(IFBPtr) do begin
    BTGetRec(IFBPtr, RecNum, Rec, False); {Read each record}
    if Rec.Dele = 0 then {Make sure it's not deleted}
      if not PutElement(Rec) then begin {Pass record to MSORT unit}
        BTCloseFileBlock(IFBPtr); {Close up if sort error}
        Exit;
      end;
    end;
    end;
  BTCloseFileBlock(IFBPtr); {Close up}
end;
```

```

procedure PutElem;
  {-Return each sorted element from the Msort unit}
var
  RecNum : LongInt;
  KeyNum : Integer;
  KeyStr : IsamKeyStr;
  IFBPtr : IsamFileBlockPtr;
  IID : IsamIndDescr;
  Rec : MyRecType;
begin
  {Initialize index descriptor, IID}
  ...
  {Recreate the fileblock}
  BTPCreateFileBlock(DataName, SizeOf(MyRecType), NumberOfKeys, IID);
  BTPOpenFileBlock(IFBPtr, DataName, False, False, False, False);
  while GetElement(Rec) do begin {Get each sorted element}
    BTAddRec(IFBPtr, RecNum, Rec); {Add the record}
    for KeyNum := 1 to NumberOfKeys do begin
      KeyStr := ... {Build the key string}
      BTAddKey(IFBPtr, KeyNum, RecNum, KeyStr); {Add the key}
    end;
  end;
  BTPCloseFileBlock(IFBPtr); {Close up}
end;

function Less(var X,Y) : Boolean;
  {-Compare elements being sorted}
begin
  {Return true if MyRecType(X) < MyRecType(Y)}
end;
{$F-}

```

Note that GetElem skips record 0, which holds internal data describing the fileblock. GetElem reads all the remaining records, but does not pass deleted records to MSORT, since sorting deleted records would be a waste of time. PutElem creates a new fileblock, in this case overwriting the old one (which probably isn't a good idea in a real application). It then gets each record back in sorted order from MSORT and adds it and its keys to the new fileblock. The Less function works like any other MSORT Less function, since deleted records have already been filtered out.

Although this example shows an outline of the required steps, it does not include error checking. Each call to a B-Tree Filer routine should be followed by a check of <IsamOK>.

MSORT's Memory Management

MSORT makes use of a special unit called TPALLOC, supplied with B-Tree Filer. TPALLOC contains special heap management routines that allow the allocation of data structures greater than 64K bytes. Structures this large may be needed by the sort system to manage internal data. Specifically, MSORT uses one structure called

the "run buffer" for an in-memory sort of the largest group of elements that will fit at one time.

By default, the run buffer is allocated as a large contiguous array (which may easily exceed 64K bytes). This speeds up the allocation and deallocation of this buffer, and simplifies the code required to manage it. There is one potential disadvantage to this technique: if the heap is fragmented, there may not be a single block of heap space large enough to allocate the run buffer contiguously.

Should this fragmentation issue become a problem, you can change MSORT's memory allocation behavior. To do so, remove the compilation directive `BigHeap` found near the top of `MSORT.PAS` and recompile the unit. Fragmentation will no longer be an issue, but a new constraint will apply. In order to avoid the time required to deallocate potentially thousands of pointers, MSORT will use a special kind of Mark and Release. This technique works correctly only if two conditions are met:

1. the caller of the MSORT routines does not perform heap allocation and deallocation while the sort is in progress--that is, within the user-defined procedures of the sort.
2. the heap is not fragmented or Turbo Pascal 6 is not used. The special Release routine used within MSORT will not work with the Turbo Pascal 6 heap manager when the heap is fragmented.

If the application needs to allocate heap space while the sort is in progress, it must take one precaution in addition to leaving the `BigHeap` directive defined. The `<MaxHeapToUse>` typed constant described below informs MSORT of how much heap space it can use while sorting. The default value allows MSORT to use all available heap space. The application must set `<MaxHeapToUse>` to a smaller value prior to calling `<DoSort>` or `<AutoSort>` in order to guarantee that heap space will be available during the sort.

MSORT also uses EMS memory if it is available and needed. (See `<AutoSort>` for more information on how it determines whether EMS will be used.) If an application uses EMS memory itself, it should allocate any pages it needs prior to calling the sort routines. The user-defined `Get`, `Put`, and `Less` routines may use EMS themselves, even transferring data from EMS directly to MSORT. MSORT also sets its mapping context every time it reads or writes EMS, so the user routines need not save and restore this context themselves. MSORT will automatically free any EMS pages it uses upon completion of the sort, or in its exit procedure if a runtime error occurs during the sort process. EMS usage can be disabled by setting the `<UseEms>` constant `False`.

MSORT's Disk Usage

When there are too many elements to sort in RAM, MSORT creates temporary files to hold the overflow. It stores these files directly in EMS if possible; otherwise, it writes them to disk. A parameter, <TempPath>, is passed to the sort functions, <DoSort> and <AutoSort>, to specify the drive and directory for these files. Of course, disk space sufficient to hold the temporary files must be available on the specified drive.

To help determine the resources required to complete a sort, a procedure called <AutoSortInfo> has been provided. <AutoSortInfo> is called automatically by <AutoSort> to determine that enough disk space exists before attempting the merge sort.

MSORT Identifiers

This section describes all identifiers exported by the MSORT unit.

Two terms that will be used throughout the discussion of MSORT are "run" and "merge order". A run is the number of items that will fit into RAM at a time. The merge order is the maximum number of files that will be used as input during the "merge phase". The "merge phase" is a process required only if the elements to be sorted will not fit into a single run.

Constants

`BiggestDataItem = 65521;`

The size of the largest single data item Turbo Pascal can handle.

`MaxHeapToUse : LongInt = 655210;`

Specifies the maximum amount of heap space that MSORT can use.

`MergeOrder = 5;`

Specifies the merge order for the sort. It controls the number of files opened during the merge phase, and affects the performance of the sort. In the worst case, there will be <MergeOrder> files open for input, and one file open for output. The default value of 5 has been carefully chosen to balance performance, memory usage, and file handle usage. You can reduce its value to as low as 2.

`STemp : String[5] = 'STEMP';`

Used as the first five characters of each temporary file name created during the merge phase.

`UseEMS : Boolean = True;`

Controls whether the merge sort system will attempt to use EMS if it is available and needed.

Types

```
MSortStatus = (MSortSuccess,      {Successful sort}  
               MSortOutOfMemory, {Insufficient memory}  
               MSortDiskError,    {Disk I/O error}  
               MSortOutOfDisk,    {Insufficient disk space for merge}  
               MSortEMSError,     {EMS error}  
               MSortUserAbort);  {User abort}
```

Both <AutoSort> and <DoSort> return a result of type <MSortStatus>. The calling application should check this result to determine the outcome of the sort. <MSortIOResult>, described below, provides additional information in case of a disk I/O error.

```
PathName = String[79];
```

String type used for pathnames.

Variables

```
GRunLength : Word;
```

Holds the run length, which is the number of items (records) that can fit in memory at one time. If all the items to be sorted fit into memory, then this will be equal to the total number of items. When using <AutoSort>, this number is calculated automatically based on available memory.

```
LastFileName : PathName;
```

Contains the pathname of the last file written to by the merge sort system.

```
MSortIOResult : Integer;
```

If the sort returns a status of <MSortDiskError>, then <MSortIOResult> will contain the value of IOResult at the time of the error.

```
UsingEMS : Boolean;
```

This variable will be True if the merge sort system is using EMS memory. EMS memory will be used only if the typed constant <UseEMS> is True, if there are enough EMS pages available for use (at least one run's worth), and if a merge phase is required.

AbortSort

Declaration

procedure AbortSort;

Purpose

Prematurely halt the sort.

Description

Any of the three user routines may call this routine to halt the sort prematurely. <DoSort> and <AutoSort> will exit at the next opportunity, returning <MSortUserAbort> in the function result.

Declaration

```
function AutoSort(FSizeInRecs : LongInt;  
                  RecLength : Word;  
                  TempPath : PathName;  
                  GetElements : Pointer;  
                  LessFunc : Pointer;  
                  PutElements : Pointer) : MSortStatus;
```

Purpose

Sort a data set.

Description

<AutoSort> is the routine you'll probably want to call to perform a sort. It works at the highest possible level, optimizing the various merge sort parameters to take advantage of system resources. <AutoSort> is designed to be completely configurable at runtime. Within the limitations of available disk space, it can sort any number of elements of any type, calling user-defined functions for element comparison, input and output. If possible, all the elements to be sorted are held in memory; otherwise, a merge sort is performed on the elements.

The first two parameters to <AutoSort> tell it how many elements to expect and the size of each element. <AutoSort> needs this information in order to accurately calculate various merge sort parameters. While <FSizeInRecs> need not be exact, it should be very close to the actual number of records sorted, or the sort may run out of resources. If the number of records to be sorted is not known in advance, use <DoSort> instead of <AutoSort>.

The <TempPath> parameter tells <AutoSort> where to put any temporary files created during the merge sort phase. Specifying the empty string, "", tells <AutoSort> to use the default drive and directory. Any other string must specify the name of an existing directory. There must be sufficient disk space on the specified drive to accommodate any temporary files created by the sort and merge phase, or <AutoSort> will exit with status <MSortOutOfDisk>.

The remaining parameters to <AutoSort> are pointers to routines called during the sort. These routines must be global, must be compiled under the far model, and must have exactly the parameters described below.

<GetElements> points to a procedure like the following:

```
{ $F+ }  
procedure GetSortElements;  
begin  
    (For each element to be sorted do...)  
    if not PutElement(X) then  
        Exit;  
end;  
{ $F- }
```

AutoSort

The routine must obtain each element to be sorted and call <PutElement> with that element as a parameter. It will generally take the form of a for or while loop.

<LessFunc> points to a function that returns True if its first element is less than the second, False otherwise. Note that the parameters passed to the Less function are untyped; they may be treated as variables of a specific type by using typecasting or variables declared absolute on top of the parameters. Actually, X and Y may be declared as parameters of the actual data type as long as they are var parameters.

```
($F+)  
function Less(var X,Y) : Boolean;  
begin  
  (Return True if X < Y, False otherwise)  
end;  
($F-)
```

<PutElements> points to a procedure that retrieves the sorted elements from MSORT using <GetElement> and makes them available to the program.

```
($F+)  
procedure PutSortElements;  
begin  
  while GetElement(X) do  
    (Do something with X) ;  
end;  
($F-)
```

Using these example routines, the call to <AutoSort> might look as follows:

```
Status := AutoSort(NumItems, RecSize, '',  
  @GetSortElements, @Less, @PutSortElements);
```

<AutoSort> calls the GetSortElements routine after performing preliminary error checking and initialization tasks. GetSortElements in turn must call <PutElement> to put each element into MSORT's internal data structures. Whenever <AutoSort> needs to compare two elements to place them in sorted order, it calls your Less function. After sorting is complete, <AutoSort> calls your PutSortElements routine, which must call <GetElement> to get each element back in sorted order. When PutSortElements returns, <AutoSort> deallocates any memory it has used and returns a sort status to the original calling program. See the <MSortStatus> type for the codes <AutoSort> uses.

Example

See the introduction to this section.

See Also

AutoSortInfo

DoSort

Declaration

```
function AutoSortInfo(FSizeInRecs : LongInt;  
    RecLength : Word;  
    var HeapSpace : LongInt;  
    var DiskSpace : LongInt;  
    var FileHandles : Word;  
    var EMSPages : Word;  
    var RunLen : Word;  
    var FileBufs : Word;  
    var OutFileBufs : Word;  
    var AllInMem : Boolean) : MSortStatus;
```

Purpose

Compute the resources needed for a particular sort.

Description

<AutoSortInfo> may be called to determine the best location for temporary files, to help choose customized parameters for <DoSort>, or to predict whether the sort can succeed given the system's resources. <AutoSortInfo> will return <MSortOutOfMemory> if the sort simply cannot be performed.

<AutoSortInfo> calculates <AutoSort>'s heap space overhead, then checks to see if the amount of free heap space will allow the entire sort to fit in RAM. If not, it chooses the largest possible run size that will fit, and simulates the merge sort process to calculate the disk space, EMS pages, and file handles required during the merge.

<HeapSpace> returns the peak bytes of heap space required. <DiskSpace> returns the peak bytes of disk space. <FileHandles> returns the highest number of file handles. <EMSPages> is the peak EMS page allocation (each page is 16384 bytes). <RunLen> is the number of records to be loaded into memory at one time. <FileBufs> is the number of bytes of heap space used for input file buffers during merging. <OutFileBufs> is the number of bytes of heap space used for the output buffer during merging. <AllInMem> is True if the sort can be performed entirely in memory, i.e., no merging is required.

DoSort

Declaration

```
function DoSort(RunLength : Word;  
                RecLength : Word;  
                InFileBufMax : Word;  
                OutFileBufMax : Word;  
                TempPath : PathName;  
                GetElements : Pointer;  
                LessFunc : Pointer;  
                PutElements : Pointer) : MSortStatus;
```

Purpose

Sort a data set with control over low-level configuration.

Description

<AutoSort> itself calls this function after it determines the optimum sort parameters. Many parameters to <DoSort> are the same as those passed to <AutoSort>; we won't describe them again here. The few that differ are described next.

<RunLength> is the number of elements to be sorted in RAM at one time. It is the caller's responsibility to assure that at least <RunLength>*<RecLength> bytes of heap space are free. (Whether this space must be contiguous or not is determined by the conditional define BigHeap, described earlier.)

<InFileBufMax> and <OutFileBufMax> specify buffer sizes to be used while merging. Again, it is the caller's responsibility to assure that sufficient heap space is free for <DoSort> to allocate these buffers. Passing zero for these parameters disables buffering altogether, at significant cost to performance.

In order for output buffering to make sense, <OutFileBufMax> should be at least twice <RecLength> (and hopefully many times more than that). The value specified by <InFileBufMax> will be divided by the <MergeOrder>, and each input file will get a buffer of that size. Hence, for effective input buffering, <InFileBufMax> should equal at least twice <RecLength>*<MergeOrder> (and hopefully many times more than that).

See Also

AutoSort

Declaration

```
function GetElement(var X) : Boolean;
```

Purpose

Return each element after sorting.

Description

Your application will have the opportunity to call <GetElement> within the retrieval routine whose address you pass to <AutoSort>.

Because MSORT does not know the type of your data, <GetElement>'s var parameter is untyped. MSORT knows how many bytes to pass back; you must make sure the variable you pass is large enough.

<GetElement> returns True as long as there are further elements to retrieve. Once it returns False, the parameter X is undefined.

Example

See the examples at the beginning of this section.

Declaration

```
function PutElement(var X) : Boolean;
```

Purpose

Add an element to be sorted.

Description

Your application will have the opportunity to call <PutElement> within the loader routine whose address you pass to the <AutoSort> function.

Because MSORT does not know the type of your data, the parameter <X> is an untyped var parameter. As such, you cannot pass a constant or expression to PutElement. MSORT knows how many bytes to transfer from <X> based on the <RecLength> passed to <AutoSort> or <DoSort>.

<PutElement> returns True when an element is successfully added to the MSORT internal structure. If it returns False, an error has occurred (probably due to insufficient memory or disk space), and the application must stop adding elements and exit the loader routine.

Example

See the examples at the beginning of this section.

7. Network Utilities

The features of B-Tree Filer we've described so far let you write databases that run quickly and reliably in almost any network environment. There is often more to writing an application that really takes advantage of a network, however. First, the application should verify that it indeed does have access to a desired network. Shared support files need to be locked and unlocked. Printed reports must be managed. And direct communication between workstations opens a new realm of program functionality, especially in the emerging era of "groupware".

Keeping these needs in mind, we've provided three additional high powered units with the multi-user version of B-Tree Filer. (They are not included with the single user version.) Recognizing Novell's dominant position in PC networking today, we offer the NETWARE unit, which provides many Novell-specific capabilities. The NETBIOS unit offers access to inter-workstation message-passing routines that are supported by almost all modern networks, including Novell, 3Com, PC LAN, and PC-NET. And the SHARE unit encapsulates the file locking routines supplied with the MS-DOS 3.x SHARE utility and also some routines specific to MS-NET and PC LAN.

Three demonstration programs demonstrate what the network utilities can do. NETINFO identifies what kind of network, if any, is active and reports everything it can determine about the network. NSEND and NRECEIVE are partners: they copy a file from one workstation to another (without going through the server) by using NetBIOS session or NetWare SPX services, both of which you'll learn more about in this chapter. These demonstration programs are described in the last section of the chapter.

For a general introduction to network concepts, refer to any of the following books:

Understanding Local Area Networks

Stan Schatt, Howard W. Sams & Co, 1988.

Local Area Networks, The Second Generation

T.W. Madrone, John Wiley and Sons, 1988.

Communications and Networking for the IBM PC and Compatibles

Larry Jordan, Brady Books, Simon & Schuster, 1986.

A. NETWARE: Novell NetWare Support

Novell supplies only C language routines to access the power of their Advanced NetWare operating system. This leaves most Turbo Pascal programmers out in the cold! The NETWARE unit supplied here implements many of the NetWare application program interfaces (APIs)--services to access the functions of a NetWare local area network (LAN). The tools in the NETWARE unit access such facilities as the network print spooler, messaging services, direct station-to-station communications, NetWare SFT's Transaction Tracking Services, and others.

There are actually many more routines in the NetWare API than we've implemented. We've chosen the ones we felt would best complement the routines in B-Tree Filer. If you need others, we hope that our source code, in combination with Novell's terse documentation, will allow you to implement what you need.

The routines within this unit are specific to the NetWare environment. Some NetWare compatible operating systems (like Network-OS from CBIS) will support some of these functions, but it is unlikely that they will support all.

In writing this chapter, we have had to assume that you are familiar with the general concepts of local area networking, and that you have some knowledge of Novell's Advanced NetWare product line. Specifically, you should know what file servers, workstations, and print spoolers are. This manual makes no attempt to explain how NetWare works, nor does it explain network cabling, topologies, or theory. For further background, we recommend the following references:

Novell Programmer's Guide

Novell API Reference Volume 1

Novell API Reference Volume 2

All available directly from Novell.

Programmer's Guide to NetWare

Charles G. Rose, McGraw-Hill, ISBN 0-07-607029-8

In this section, we spell NETWARE in uppercase letters to indicate the unit provided with B-Tree Filer, while we use Novell's mixed case, NetWare, to denote their operating system.

The NETWARE unit includes tools that can be grouped into five categories:

- file and directory management
- transaction tracking
- print spooling

- basic message services (pipes and broadcasts)
- advanced message services (IPX and SPX)

In order to keep related routines together, we'll discuss each category in a separate section.

If you delve into the NETWARE source code, you may notice that some identifiers are interfaced there but not documented here. Although these identifiers all serve useful functions, we estimated that documenting them would turn this chapter of the B-Tree Filer manual into a tome the size of Novell's complete developer documentation. We've drawn the line at documenting those types and routines that are reasonably accessible to the programmer who isn't a full time network hacker. If you have Novell's documentation and want to use NETWARE's undocumented routines, go for it! We use them ourselves.

File and Directory Attributes

The first group of tools in the NETWARE unit covers file, directory, and informational services. Supplied routines offer the following functions:

- determine whether NetWare is loaded
- get or set extended file attributes
- determine whether a file is shareable or make it so
- get the NetWare handle of a path on a network drive
- get or set the directory rights (read, write, etc.)
- switch between NetWare's extended lock and DOS compatibility modes
- determine whether a workstation has console privileges
- read the network server's date and time

Constants

NwRead = \$01;	NwWrite = \$02;
NwOpen = \$04;	NwCreate = \$08;
NwDelete = \$10;	NwParental = \$20;
NwSearch = \$40;	NwModify = \$80;

Definitions of bits set in the effective rights mask returned by <GetDirMask>.

NwPermanent = \$01;
NwTemporary = \$02;
NwLocal = \$80;

Definitions of bits set in the <StatusFlags> parameter returned by <GetDirHandle>.

Types

DayOfTheWeek = (Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday);

Used in conjunction with the <GetServerDateTime> routine.

```
DriverInformation =  
  record  
    NetworkAddress : LongInt;  
    HostAddress : PhysicalNodeAddress;  
    OptionNumber : Byte;  
    LanBoardName : Str80;  
    HardwareSettings : Str80;  
  end;
```

Describes the data structure returned by the <GetDriverConfiguration> routine. This type returns information about the LAN adapter installed in the default server. <NetworkAddress> is the Internet network number of the current network. <HostAddress> is the physical node address of the server. The other fields return

status information regarding the type of adapter (e.g., Ethernet) and hardware settings (e.g., IRQ=9, IO Address=300h) of the adapter.

```
ServerInformation =  
  record  
    Len : Word;  
    ServName : array[1..48] of Char;  
    NetWareVer : Byte;  
    NetWareSub : Byte;  
    MaxConns : Word;  
    UsedConns : Word;  
    MaxVols : Word;  
    Revision : Byte;  
    SFTLevel : Byte;  
    TTSLevel : Byte;  
    PeakConn : Word;  
    AccountVer : Byte;  
    VAPVer : Byte;  
    QueueVer : Byte;  
    PrintServVer : Byte;  
    VirtualVer : Byte;  
    SecurityVer : Byte;  
    BridgeVer : Byte;  
    Reserved : array[1..60] of Byte;  
  end;
```

A structure returned by the procedure <GetServerInfo>. Most of the fields describe version numbers of the NetWare software running on the server. The most important fields are <NetWareVer> (the major version number), <NetWareSub> (the minor version number), <MaxConns> (the maximum number of workstation connections), and <UsedConns> (the number of connections in use).

ConsolePriv

Declaration

```
function ConsolePriv : Boolean;
```

Purpose

Returns True if station has console privileges.

Description

Certain NetWare services require that the caller have console privileges. Of those included in the NETWARE unit, only <GetDriverConfiguration>, <TTSDisable>, and <TTSEnable> require that privilege. This routine determines whether such privileges belong to the caller.

Example

```
if ConsolePriv then  
  Writeln('Workstation has console privileges')  
else  
  Writeln('Workstation has no console privileges');
```

See Also

NetWareLoaded

TTSDisable

Declaration

```
function FileIsShareable(Path : PathName;  
    var FAttr : Word;  
    var ErrCode : Word) : Boolean;
```

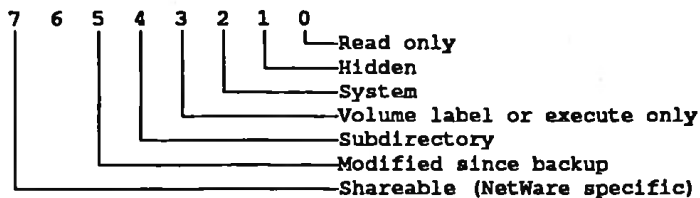
Purpose

Return True if <Path> is shareable.

Description

The file's DOS attributes are returned in <FAttr>.

The DOS file attributes are a bit-mapped word, interpreted as follows:



Actually, the attribute is a byte, but a word is chosen for consistency with Turbo Pascal's DOS unit.

NetWare uses the high bit of the normal DOS file attribute byte to indicate that a file is shareable. In order for more than one workstation to access a file, this bit must be explicitly set when the file is created, or later by an application.

Errors returned in the <ErrCode> parameter include:

- 0 Success.
- 2 File not found.
- 5 Access denied.

Example

```
Shareable := FileIsShareable('ADDRESS.DAT', Attr, ErrCode);  
if ErrorCode = 0 then begin  
    if Shareable then  
        Writeln('ADDRESS.DAT is shareable')  
    else  
        Writeln('ADDRESS.DAT is not shareable');  
end else  
    Writeln('ADDRESS.DAT not found or access denied');
```

Reports the shareable status of ADDRESS.DAT.

See Also

MakeFileShareable

GetExtFAttr

GetDirHandle

Declaration

```
function GetDirHandle(Drive : Char; var StatusFlags : Byte) : Byte;
```

Purpose

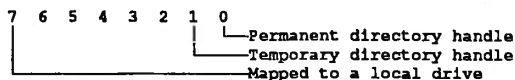
Determine the NetWare directory handle associated with a drive.

Description

<Drive> specifies a drive letter, e.g., 'A', 'B', etc. (Case is not important.) Many NetWare file services do not deal directly with disk drives; instead they use a single byte directory handle.

If the function result returns 0, it indicates that the specified <Drive> is invalid. Otherwise the value is the handle for <Drive>.

The status flags are a bit-mapped byte interpreted as follows:



Permanent directory handles are not deallocated when the current program ends, while temporary handles are. "Mapped to a local drive" means that the directory handle refers to a drive physically located on the workstation. If none of the flags is set, the drive is not mapped to a directory handle.

Example

```
var
  DirHandle, Status : Byte;
...
DirHandle := GetDirHandle('J', Status);
Write('Drive J: ');
if Status = 0 then
  Writeln('is not mapped to a directory handle');
else begin
  Write('Handle: ', DirHandle);
  if (Status and NWLocal) <> 0 then Write(' local drive');
  if (Status and NWPermanent) <> 0 then Write(' permanent dir handle');
  if (Status and NWTemporary) <> 0 then Write(' temporary dir handle');
  Writeln;
end;
```

Reports on the directory handle of drive J.

See Also

GetDirPath

GetDirRights

Declaration

```
function GetDirPath(DirHandle : Byte; var Path : String) : Byte;
```

Purpose

Returns the current directory for the specified drive handle.

Description

The function returns 0 if successful, or \$9B to indicate a bad directory handle.

<Path> is of type String because NetWare allows for longer paths than DOS. The returned string can actually be up to 255 bytes long.

Example

```
var
  Path : String;
...
if GetDirPath(1, Path) = 0 then
  Writeln('Current directory for handle 1 is ', Path)
else
  Writeln('1 is not a valid handle');
```

Displays the current directory for handle 1.

See Also

GetDirHandle

GetDirRights

GetDirRights

Declaration

```
function GetDirRights(DirHandle : Byte;  
    Path : String;  
    var Rights : Byte) : Byte;
```

Purpose

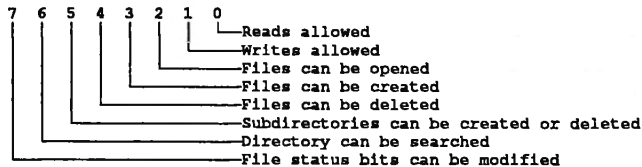
Returns workstation rights for the specified <Path> of drive <DirHandle>.

Description

The effective directory rights determine what file operations the station may perform in the specified directory. <DirHandle> must have been previously determined by calling <GetDirHandle>. <Path> should not include a drive letter. The subdirectory is interpreted relative to the drive and current directory specified by <DirHandle>.

<Path> is of type String because NetWare allows for longer paths than DOS.

<Rights> is a bit-mapped byte interpreted as follows:



Error codes that may be returned in the function result include:

\$00 Success.
\$98 Volume does not exist.
\$9B Bad directory handle.

Example

```
if GetDirRights(GetDirHandle('J', HandleStatus),  
    '\LOGIN', RightsStatus) = 0 then  
    Writeln('Workstation rights to J:\LOGIN directory ', RightsStatus);  
Reports the status byte for the current workstation's rights in J:\LOGIN.
```

See Also

GetDirHandle

GetDirPath

Declaration

```
function GetDriverConfiguration(LanBoard : Byte;  
                               var Driver : DriverInformation) : Boolean;
```

Purpose

Returns information about the LAN driver and adapter installed in the default server.

Description

The first adapter in the server is number 0. The caller must have console privileges in order for this routine to return True.

See type <DriverInformation> near the beginning of this section for more information.

Example

```
var  
  DI : DriverInformation;  
...  
  if GetDriverConfiguration(0, DI) then  
    Writeln('Server adapter: ', DI.LanBoardName)  
  else  
    Writeln('Unable to get driver configuration');
```

Displays the transport hardware type (e.g., Ethernet, Token Ring) for the server's default LAN adapter card.

See Also

GetServerInfo

frequently accessed files larger than 2 megabytes. Read and write audit bits indicate that accesses to the file will be recorded in an audit file (a feature not implemented in some versions of NetWare).

Status codes are returned in function result:

- 0 Success.
- 2 File not found.
- 18 Requesting station does not have search rights.

Example

```
var
  Attr : Byte;
...
if GetExtFAttr('TEST.DAT', Attr) = 0 then begin
  Attr := Attr or $10;
  if SetExtFAttr('TEST.DAT', Attr) = 0 then
    Writeln('TEST.DAT now allows transaction tracking');
end;
```

Gets the current extended attributes for file TEST.DAT and then sets the transaction tracking bit while maintaining other extended attributes.

See Also

SetExtFAttr

TTSBegin

GetServerDateTime

Declaration

```
procedure GetServerDateTime(var Year : Word;  
                           var Month, Day, Hour, Minute, Second : Byte;  
                           var WeekDay : DayOfTheWeek);
```

Purpose

Returns the network time and date from the file server.

Description

Parameters returned by this routine are interpreted as follows:

Year	1980..2155
Month	1..12
Day	1..31
Hour	0..23
Minute	0..59
Second	0..59
Weekday	Sunday..Saturday

Since workstations running under NetWare may have different internal clock settings, it is good practice to synchronize network operations using the time and date returned by the server. Whether the application should forcibly reset the local date/time or just ignore it is up to the programmer.

Example

```
GetServerDateTime(Year, Month, Day, Hour, Minute, Second, WeekDay);  
WriteLn('Date on the file server = ', Month, '/', Day, '/', Year);
```

Displays the date according to the file server.

See Also

GetServerInfo

Declaration

```
procedure GetServerInfo(var ServerInfo : ServerInformation);
```

Purpose

Returns detailed information about the default file server.

Description

Information includes the version of NetWare running on the server, the server's name, the maximum number of connections, and the number of connections currently in use. See type <ServerInformation> for more information about the fields in <ServerInfo>.

See Also

GetServerDateTime

GetWSInfo

Declaration

```
procedure GetWSInfo(var ShellMajor, ShellMinor, ShellRevision : Byte;  
var OSType, OSVer, Hardware : Str10);
```

Purpose

Returns NetWare version information from the local copy of the shell.

Description

The parameters returned by this procedure are interpreted as follows:

ShellMajor	Integer portion of the NetWare shell version
ShellMinor	Fractional portion
ShellRevision	Shell revision number
OSType	Type of NetWare version
OSVer	String representation of the version number
Hardware	Hardware type being used by the network

This function is available only for Advanced NetWare versions 2.10 or later.

<GetServerInfo> is available for all versions of NetWare. <GetWSInfo> returns zeros or null parameters if the function is not supported.

See Also

GetServerInfo

IsLockModeExtended

Declaration

```
function IsLockModeExtended : Boolean;
```

Purpose

Returns True if using Advanced NetWare extended lock mode.

Description

NetWare locks operate in either of two modes: extended or compatibility. Compatibility mode is the default and should be used if the application is to run on NetWare versions prior to 4.61.

Example

```
Write('Lock mode: ');  
if IsLockModeExtended then  
  Writeln('Extended')  
else  
  Writeln('Compatibility');
```

Displays the current lock mode.

See Also

SetLockMode

MakeFileShareable / NetWareLoaded

Declaration

```
function MakeFileShareable(Path : PathName) : Byte;
```

Purpose

Marks the file specified in <Path> as shareable.

Description

This sets the high bit of the file attribute, but does not modify any other bits. For example, using this routine on a read-only hidden file would result in a shareable read-only hidden file.

A status code is returned in function result:

```
0   Success.
2   File not found.
5   Access denied.
```

Example

```
if MakeFileShareable('TEST.DAT') = 0 then
  Writeln('File TEST.DAT is now shareable');
```

Marks TEST.DAT shareable and reports on the outcome.

See Also

FileIsShareable

Declaration

```
function NetWareLoaded(var LoggedOn : Boolean) : Boolean;
```

Purpose

Determine whether station has shell loaded and is logged on to a NetWare server.

Description

Applications that need to access NetWare services must call this routine first to assure that the NetWare shell (e.g., NET3.COM, ANET3.COM, NET4.COM, EMSNET3.EXE) has been loaded on the current workstation. The <LoggedOn> parameter returns whether the current station has logged on to the network. Both of these are required in order for the application to successfully use the routines in the NETWARE unit.

Example

```
if NetWareLoaded(Logged) then begin
  if Logged then
    Writeln('Logged onto server')
  end else
    Writeln('Not logged onto server');
end else
  Writeln('NetWare shell not loaded');
```

SetExtFAttr / SetLockMode

Declaration

```
function SetExtFAttr(Path : PathName; Attr : Byte) : Byte;
```

Purpose

Sets the extended attributes of the file <Path> to the value <Attr>.

Description

See <GetExtFAttr> for more information about the possible attributes.

A status code is returned in function result:

- 0 Success.
- 2 File not found.
- 5 Access denied.

Example

See the example for <GetExtFAttr>.

Declaration

```
procedure SetLockMode(Extended : Boolean);
```

Purpose

Sets the lock mode.

Description

Pass <Extended> = True to enable extended locking mode. By default, locking works in compatibility mode, which should be used if the application is to run on NetWare versions prior to 4.61.

Example

```
SetLockMode(True);  
Enable extended locking.
```

See Also

IsLockModeExtended

Transaction Tracking

Novell's Advanced NetWare SFT (system fault tolerant NetWare) provides a sophisticated transaction tracking system to improve data integrity. With transaction tracking services (TTS), the network operating system can guarantee that a complete sequence of operations is written to disk, or that none of it is. The NETWARE unit supports these functions as well:

- determine whether TTS services are available
- begin, end, or abort a transaction
- check status of a transaction
- disable or enable transaction tracking

NetWare supports two categories of transactions: explicit and implicit. Explicit transactions are initiated when the application makes an explicit call to the operating system. Implicit transactions are started automatically when the operating system recognizes that certain levels of record locking have been activated by the application. We've documented only the explicit method here; for more information on implicit transactions, see the Novell documentation and NETWARE's procedures <TTSGetAppThresh> and <TTSGetWSThresh>.

Examples

The most common time to use TTS is while adding, deleting, or modifying a database record. TTS is valuable in this case because it can guarantee that the entire transaction--updating the data file and all associated indexes--will occur completely or not at all. Even B-Tree Filer's save mode cannot guarantee this by itself. The following example, which builds on the routines discussed in Chapter 4, shows an outline of the steps required to use TTS in this way.

```
var
  UseTTS : Boolean;
  TransactionActive : Boolean;

procedure BeginTransaction;
begin
  if UseTTS and not TransactionActive then
    case TTSBegin of
      0, $FE, $FF : TransactionActive := True;
    else
      {Out of workspace. Error handling}
    end;
end;
```

```

procedure EndTransaction;
var
  ID : LongInt;
  Retries : Word;
begin
  if UseTTS and TransactionActive then
    case TTSEnd(ID) of
      0, $FE :
        begin
          Retries := 0;
          while Retries < 100 do begin
            if TTSStatus(ID) then
              {Transaction committed to disk}
              Exit;
            Delay(100);
            inc(Retries);
          end;
          {Transaction lost. Error handling}
        end;
      else
        {TTS Disabled. Error handling}
      end;
    end;
end;

procedure AbortTransaction;
begin
  if UseTTS and TransactionActive then
    case TTSAbsort of
      0, $FE, $FF : TransactionActive := False;
    else
      {TTS Disabled. Error handling}
    end;
end;

function AddRecord(P : PersonDef) : Boolean;
var
  KeyNr : Integer;
  RefNr : LongInt;
  Key : IsamKeyStr;
begin
  AddRecord := False;

  {Lock the database}
  repeat
    BTLockFileBlock(PF);
  until not IsLockError(True);

  {Begin a transaction}
  BeginTransaction;

  BTAddRec(PF, RefNr, P);
  if not IsamOK then begin
    {Error handling}
    BTUnlockFileBlock(PF);
    AbortTransaction;
    Exit;
  end;
end;

```

```

    for KeyNr := 1 to BTNrOfKeys(PF) do begin
        Key := CreateKey(P, KeyNr);
        BTAddKey(PF, KeyNr, RefNr, Key);
        if not IsamOK then begin
            {Remove keys added so far}
            UndoAdd(P, RefNr, KeyNr-1);
            {Remove the new record}
            BTDeleteRec(PF, RefNr);
            {Error handling}
            BTUnlockFileBlock(PF);
            AbortTransaction;
            Exit;
        end;
    end;

    BTUnlockFileBlock(PF);
    if IsamOK then
        EndTransaction
    else
        AbortTransaction;
    AddRecord := IsamOK;
end;

var
    LoggedOn : Boolean;

begin
    {Assure that TTS is available}
    UseTTS := False;
    if NetWareLoaded(LoggedOn) then
        if LoggedOn then
            UseTTS := TTSAvailable;
        TransactionActive := False;
    ...
end.

```

Encapsulating the TTS calls as shown simplifies the rest of the program logic. The example also shows how to defeat attempts to start a new transaction while another is already active.

Note that the EndTransaction procedure does not return until the transaction has actually been committed to disk. (NetWare may cache disk data in memory for several seconds.) EndTransaction's conservative approach is not always recommended because it may lead to unpleasant delays in program response. Another approach would be to maintain a list of the ID's whose transactions have ended and to call <TTSSStatus> later during execution. This approach has its drawbacks as well--if several seconds have elapsed and the transaction has still not been committed, what should the application do? Such a result probably means that the server has crashed or the network has gone down. About the only reasonable response is to report the uncommitted ID numbers to the user (so they can be re-entered later) and halt the application.

If an error occurs the AddRecord routine must perform the same "undo" actions that it does when TTS is not in use. These actions update B-Tree Filer's own internal buffers; AbortTransaction guarantees that none of the updates appear on disk.

Finally note that the application must determine whether TTS is available before attempting to use it. Another important step is not shown. Before transaction tracking can be used on any file, Novell's extended file attribute that enables tracking must be assigned to the file. See <SetExtFAttr> earlier in this chapter for more information. For a B-Tree Filer fileblock, the appropriate attribute must be assigned to the data, index, and dialog files.

TTSAbort / TTSAvailable

Declaration

function TTSAbort : Byte;

Purpose

Aborts any pending transaction.

Description

If <TTSAbort> returns successfully, the transaction has been "backed out", meaning that any disk updates associated with the transaction are undone or removed from memory buffers. In order for this function to work, TTS must be available and enabled.

Aborting a transaction usually releases all physical and logical record locks related to the transaction.

A status code is returned in function result:

- \$00 Transaction successfully aborted.
- \$FD TTS is disabled (no backout was performed).
- \$FE Transaction aborted but records remain locked.
- \$FF No explicit transaction was active.

See Also

TTSTBegin

TTSEnd

Declaration

function TTSAvailable : Boolean;

Purpose

Determine whether TTS services are available.

Description

Call this function prior to any other TTS routines to determine whether Transaction Tracking Services are available.

See Also

TTSTDisable

TTSEnable

TTSBegin

Declaration

function TTSBegin : Byte;

Purpose

Begins an explicit transaction.

Description

A "transaction" is a sequence of logically related file operations that must happen as a group--either all must be successful or none should occur. After the transaction begins, no related file operations are committed to disk until <TTSEnd> is called.

In order for a file to be used with transaction tracking, it must be flagged as transactional (see <GetExtFAttr>). Also, TTS services must be available and enabled (see <TTSAvailable> and <TTSEnable>).

A status code is returned in function result:

\$00 Transaction begun.

\$96 Out of dynamic workspace.

\$FE Implicit transaction already active (implicit now made explicit).

\$FF Explicit transaction already active (existing transaction continues).

Only \$96 is considered an error condition. If possible, the application should try again later.

Implicit transactions, mentioned in error \$FE, are not described here. See Novell's API Reference, Volume 2, for more information.

See Also

TTSAbort

TTSEnd

TTSDisable / TTSEnable

Declaration

```
function TTSDisable : Boolean;
```

Purpose

Disable TTS services network-wide.

Description

The caller must have console privileges in order to disable TTS (see <ConsolePriv>). The function returns True if it succeeded; False otherwise.

Any transactions written after TTS has been disabled cannot be undone. A transaction that has not written anything since transactions were disabled can still be reversed while transactions are disabled. Normally, transactions should be disabled only for the purposes of testing.

See Also

ConsolePriv

TTSEnable

Declaration

```
function TTSEnable : Boolean;
```

Purpose

Enable TTS services.

Description

The caller must have console privileges in order to enable TTS (see <ConsolePriv>). If TTS is available, it is active by default. The function returns True if it could enable TTS; False otherwise.

Any previous transaction backout information is erased, except for transactions that were active at the time transactions were disabled and did not write anything thereafter. These transactions are not erased and can still be backed out.

Always verify that TTS services are available (by calling <TTSAvailable>) before issuing <TTSEnable>. NetWare may report True for <TTSEnable> if TTS is not available.

See Also

ConsolePriv

TTSDisable

TTSEnd

Declaration

function TTSEnd(var ID : LongInt) : Byte;

Purpose

Ends a transaction.

Description

The transaction has not necessarily been committed to disk when this function returns. Use the <ID> value returned by <TTSEnd> in a call to <TTSSStatus> to determine when a transaction has been written to disk.

If the server crashes before a transaction has been fully committed to disk, any changes will be undone when the file server is rebooted.

If transaction tracking services are disabled but available, <ID> can still be passed to <TTSSStatus> to determine when the transaction has been committed to disk. Any transactions written after transactions have been disabled cannot be backed out.

Ending a transaction releases all physical and logical record locks related to the transaction.

The function returns a status code:

\$00 Transaction ended successfully.

\$FD TTS disabled.

\$FE Transaction ended but records remain locked.

\$FF No explicit transaction was active.

See Also

TTSEBegin

TTSSStatus

Declaration

```
function TTSStatus(ID : LongInt) : Boolean;
```

Purpose

Determine whether a transaction has been committed to disk.

Description

Call <TTSStatus> after calling <TTSEnd> to verify that a transaction has been committed to disk. <ID> is the transaction reference number returned by <TTSEnd>. The function returns True if the function has been committed; False if it hasn't yet been committed.

Do not wait for transactions to be written to disk unless it is absolutely necessary. NetWare's file server caching algorithms may result in a 3 to 5 second delay before they are actually written.

This function may also be used to determine if a transaction has been committed to disk even if <TTSEnd> reports that transaction tracking has been disabled.

See Also

TTSDisable

TTSEnd

Print Spooling

The next set of routines in the NETWARE unit controls the network print spooling functions. A few definitions are in order, since the term "spooling" has taken on several meanings in the PC community. When a NetWare workstation activates spooling, all output that would normally go to the default local printer (LPT1, for example) is instead routed to a "capture" file on the file server. As far as an application can tell, printing then proceeds as usual, even though the output is simply appending to the capture file. Three actions are then possible. The application can terminate spooling normally, in which case NetWare closes the capture file and adds this file to a queue of jobs that will be printed on the server's printer. The application can "flush the spooler", which closes the current capture file, submits that file to the print queue, and opens another capture file where spooling continues. Or the application can abort the spooler, which closes and deletes the capture file, and restores printing to the local printer.

To support these operations, NETWARE includes the following capabilities:

- identify or change the default local printer
- enable or disable the print spooler
- flush the spooler to the printer
- determine whether spooling is active
- get printer status
- control banner pages on print jobs

Examples

In addition to providing printer access to stations that don't have a printer attached, spooling improves application performance by buffering print output to disk instead of to the slow printer device. The following example shows how to use NetWare print spooling services.

```
var
  LPT : Text;
  PrintJob : PrintJobType;
  CaptureStatus : Byte;
begin
  {Open Turbo Pascal access to the printer}
  Assign(LPT, 'LPT1');
  Rewrite(LPT);

  {Assure NetWare will capture LPT1 output}
  SetDefaultLocalPrinter(LPT1);

  {Define characteristics of the printout}
  GetPrintJobFlags(PrintJob);
```

```

(Modify any desired fields in the PrintJob)
***
SetPrintJobFlags(PrintJob);

{Start spooling}
CaptureStatus := StartLPTCapture;
if CaptureStatus <> 0 then
  {Error handling} ;

{Write output to LPT1, captured by NetWare}
WriteLn(LPT, 'hello world');

{Submit what's been printed so far to the print queue}
Flush(LPT);
CaptureStatus := FlushLPTCapture;

{Printer output will still be spooled}
WriteLn(LPT, 'goodbye world');

{Submit further output to the print queue, and end capture}
Close(LPT);
CaptureStatus := EndLPTCapture;
end.

```

This example explicitly opens the local printer LPT1 as a text file. It would work just as well to Use Borland's PRINTER unit in the program and write to the Lst file.

The call to <SetPrintJobFlags> allows the application to control various characteristics of the printout, e.g., which server printer will be used, whether a banner page will appear, and the number of copies to print.

Note the calls to Turbo Pascal's Flush and Close routines. The application must call these prior to calling the NetWare FlushLPTCapture and EndLPTCapture routines to assure that Turbo's internal buffers are written out.

The example does not show the calls to assure that NetWare is loaded. Of course, this would be necessary in a real program.

Types

```
PrinterDevice = (LPT1, LPT2, LPT3);
```

Represents any of the three printer ports. The ordinal value of each element is the value used by the BIOS to represent the printer.

```
PrintJobType =  
  record  
    Status : Byte;  
    Flags : Byte;  
    TabSize : Byte;  
    ServerPrinter : Byte;  
    NumCopies : Byte;  
    FormType : Byte;  
    Reserved : Byte;  
    Banner : array[1..13] of Char;  
    LocalLPT : Byte;  
    FlushCaptureTimeout : Word;  
    FlushCaptureOnClose : Byte;  
  end;
```

Describes a print job. See function <GetPrintJobFlags> for more information.

CancelLPTCapture / EndLPTCapture

Declaration

function CancelLPTCapture : Byte;

Purpose

Aborts the capture of local print output.

Description

The capture file is not queued for printing but is instead deleted. Call <CancelLPTCapture> prior to <EndLPTCapture> or <FlushLPTCapture> to avoid printing.

To start capturing print output again, call <StartLPTCapture>.

The function returns a status code in its function result. Novell doesn't specify any particular error codes for this function, but any non-zero value indicates that an error occurred.

See Also

EndLPTCapture

StartLPTCapture

Declaration

function EndLPTCapture : Byte;

Purpose

Ends the capture of print output.

Description

The capture file is closed and the file is queued for physical printing on the default printer of the server. To abort the print job, call <CancelLPTCapture> instead.

To start capturing print output again, call <StartLPTCapture>.

The function returns a status code in its function result. Novell doesn't specify any particular error codes for this function, but any non-zero value indicates that an error occurred.

See Also

CancelLPTCapture

StartLPTCapture

FlushLPTCapture / GetBannerUser

Declaration

```
function FlushLPTCapture : Byte;
```

Purpose

Flush spooled output to printer and continue spooling.

Description

Adds the current capture file to the server's print queue and starts a new capture file. Do not call until <StartLPTCapture> has been called.

The function returns a status code in its function result. Novell doesn't specify any particular error codes for this function, but any non-zero value indicates that an error occurred.

See Also

CancelLPTCapture EndLPTCapture

Declaration

```
function GetBannerUser(var BannerUser : Str20) : Boolean;
```

Purpose

Return the name appearing on print job banners.

Description

When the NetWare spooling facilities are used, a banner page that identifies the origin of each print job precedes the printout itself. <GetBannerUser> returns the user name that will appear on this banner page. The user name may be changed by calling <SetBannerUser>. These functions are supported only by Advanced NetWare version 2.1 or later. The function returns True if the function is supported; False otherwise.

Example

```
var
  Banner : Str20;
...
  if GetBannerUser(Banner) then
    Writeln('The banner name is ', Banner)
  else
    Writeln('The banner name cannot be determined');
```

Reports the current station's banner name if possible.

See Also

SetBannerUser

GetDefaultLocalPrinter / GetPrinterStatus

Declaration

```
function GetDefaultLocalPrinter : PrinterDevice;
```

Purpose

Returns which local LPT device is available for capture.

Description

The printer-related routines in the NETWARE unit all work with the default local printer. This routine determines which printer is currently the NetWare default. The default printer may be changed by calling <SetDefaultLocalPrinter>.

Example

```
Write('Capture for LPT');  
case GetDefaultLocalPrinter of  
  LPT1 : Write('1');  
  LPT2 : Write('2');  
  LPT3 : Write('3');  
end;  
Writeln(' is active');
```

Assuming capture has started, writes the local printer port that has been captured.

See Also

SetDefaultLocalPrinter

Declaration

```
procedure GetPrinterStatus(ServerPrinter : Byte;  
                           var PrinterNo, FormType : Byte;  
                           var OffLine, Stopped : Boolean);
```

Purpose

Returns assorted information about the specified server printer.

Description

The parameters are interpreted as follows:

ServerPrinter	Network printer number (0..4).
PrinterNo	Returns same as ServerPrinter unless rerouted at server.
FormType	Form type currently selected for this printer (0..255).
OffLine	True if printer is offline.
Stopped	True if printer has been stopped at the server.

Note that <ServerPrinter> refers to the physical printers attached to the server and not to any local printers attached to the current workstation. The procedure <GetPrintJobFlags> will return more detailed information about the printer. See that procedure for additional information.

See Also

GetPrintJobFlags

GetPrintJobFlags / LPTCaptureActive

Declaration

```
procedure GetPrintJobFlags(var PrintJob : PrintJobType);
```

Purpose

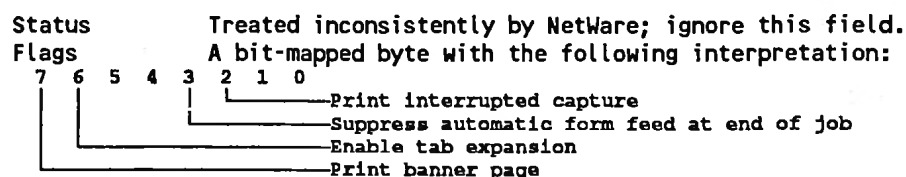
Return printing job flags for current workstation.

Description

The job flags control the various options related to network printing.

<GetPrintJobFlags> fills in the <PrintJob> variable with the setting of print job flags for the current workstation. Call <SetPrintJobFlags> to change printing options.

Lots of information is returned, some of it fairly esoteric. The following fields are of general interest:



The default value for flags is 80h, causing a banner page to be printed.

TabSize Number of spaces for each tab (1..18).
ServerPrinter Server printer used for output (0..4).
NumCopies Number of copies to print.
FormType Which form is to be mounted in printer when job starts.
When the form is not the same as the one currently mounted, a message is displayed on the server console. The default form type is 0.

Banner Name to be printed on banner. If all nulls, the capture file name is printed.

LocalLPT Local printer being captured (0..2).

See Also

SetPrintJobFlags

Declaration

```
function LPTCaptureActive(var QueServer : Byte) : Boolean;
```

Purpose

Returns True if local printer output is being captured.

Description

<QueServer>, which is initialized only when the result is True, returns the connection number (1..8) of the server processing the print queue.

See Also

EndLPTCapture

StartLPTCapture

SetBannerUser / SetDefaultLocalPrinter

Declaration

```
function SetBannerUser(UserName : Str20) : Boolean;
```

Purpose

Change the print job banner name.

Description

When the NetWare spooling facilities are used, a banner page that identifies the origin of each print job precedes the printout itself. <UserName> is the name that will appear on this banner page and may be at most 12 characters long. These functions are supported only by Advanced NetWare version 2.1 or later. The function returns False if the service is not available.

See Also

GetBannerUser

Declaration

```
procedure SetDefaultLocalPrinter(LPTNo : PrinterDevice);
```

Purpose

Change the NetWare default local printer port.

Description

The printer-related routines in the NETWARE unit all work with the default local printer. This routine sets the printer whose input will be captured for spooling. <LPTNo> may have the value <LPT1>, <LPT2>, or <LPT3>.

See Also

GetDefaultLocalPrinter

SetPrintJobFlags / SpecifyCaptureFile

Declaration

```
procedure SetPrintJobFlags(var PrintJob : PrintJobType);
```

Purpose

Change various parameters of a print job.

Description

This routine allows an application to control various print job parameters such as tab size, number of copies, and form type. See <GetPrintJobFlags> for more information. Because of the large amount of information stored in a variable of type <PrintJobType>, the application should call <GetPrintJobFlags> first to initialize the record, then modify the desired fields before calling <SetPrintJobFlags>.

Note that the changed parameters affect only print jobs initiated with <StartLPTCapture>.

See Also

GetPrintJobFlags

Declaration

```
function SpecifyCaptureFile(FName : String) : Byte;
```

Purpose

Specify the file to be used for the next print capture.

Description

The capture file is opened by this call and spooling is enabled. The capture file is closed when <CancelLPTCapture>, <EndLPTCapture>, or <FlushLPTCapture> is called. The file is not automatically queued for printing, however. This must be done by explicitly calling <SpoolExistingFile>. <SpecifyCaptureFile> is useful for setting up your own print queuing routines.

The specified file must be located on the active file server, or an error \$9C will be returned. Zero is returned to indicate success.

Example

```
if SpecifyCaptureFile('J:\SPOOL\PRINTER.OUT') = 0 then
  Writeln('Spooling enabled to PRINTER.OUT')
else
  Writeln('Unable to start spooling');
```

Starts spooling to PRINTER.OUT in an existing directory on the default file server.

See Also

SpoolExistingFile

SpoolExistingFile / StartLPTCapture

Declaration

```
function SpoolExistingFile(FName : String) : Byte;
```

Purpose

Submits an existing file to the default server's print queue.

Description

This function accesses a service much like Novell's NPRINT utility. The file must be located on one of the file server's disk drives. Note that the parameters of the printout (the banner, etc.) *cannot* be changed by calling <SetPrintJobFlags>. Apparently NetWare uses a separate set of job flags for printouts made using the <SpoolExistingFile> service.

Example

```
if SpoolExistingFile('J:\SPOOL\PRINTER.OUT') <> 0 then  
  Writeln('Unable to submit PRINTER.OUT to the print queue');
```

Attempts to submit PRINTER.OUT to the print queue. If PRINTER.OUT was created by <SpecifyCaptureFile> it should be flushed and closed by calling <EndLPTCapture> first.

See Also

SpecifyCaptureFile

Declaration

```
function StartLPTCapture : Byte;
```

Purpose

Start spooling local printer output for printing on the server.

Description

After this routine is called, output to the default local printer will be routed to a file (managed by the file server). Print output will be queued for actual printing only after calling <EndLPTCapture> or <FlushLPTCapture>. Print output may be aborted by calling <CancelLPTCapture>. The spooled local printer defaults to LPT1. To determine the current local printer, call <GetDefaultLocalPrinter>. To change the default local printer, call <SetDefaultLocalPrinter>.

The function returns a status code in its function result. Novell doesn't specify any particular error codes for this function, but any non-zero value indicates that an error occurred.

See Also

CancelLPTCapture
GetDefaultLocalPrinter

EndLPTCapture
SetDefaultLocalPrinter

Basic Message Services

NetWare has the capability to send messages in a number of ways. Broadcast messages are very short strings (less than 56 characters) displayed on the receiving workstation's monitor. Message pipes provide an easy method to send short messages (less than 127 characters) from station to station using the server as an intermediary. The NETWARE unit provides the following routines to access these services:

- get connection (node) numbers of workstations and servers
- associate physical addresses with names and connections
- get or set broadcast mode (display or ignore broadcasts)
- broadcast a message
- read a pending broadcast message
- open, check, or close a message pipe
- send or receive a message via a pipe

Neither of these methods guarantees that the recipient will read the message. Novell recommends the SPX services (described later) for efficient node-to-node communication with guaranteed delivery.

Warning: Novell has dropped support for connection-based messages in NetWare 386. Any message function that starts by getting a list of connections will no longer work. Do not use these services if your applications must run on NetWare 386 networks.

Examples

The supplied program MESEXAMP.PAS demonstrates a number of the methods used to pass messages on NetWare and NetBIOS LANs. This section will introduce the NetWare broadcast and pipe methods used in that program.

Sending a message to the NetWare console (on the default file server) is the easiest method of all. The string may be at most 60 characters long.

```
BroadcastToConsole('Hello console operator!');
```

Similarly, a message may be appended to the console log file, giving it some degree of permanence:

```
LogNetworkMessage('note for the log file');
```


Messages sent by pipes and broadcasts start the same way--by generating a list of the connections (workstations) to receive the message. This can be done in several ways. If the message is to be sent to all connections, then the following sequence should be used:

```

procedure AllConnections(var Connect : ConnectionList);
var
  I : Word;
  ServerInfo : ServerInformation;
begin
  GetServerInfo(ServerInfo);
  with Connect do begin
    {Add all valid connections to the list}
    Count := ServerInfo.MaxConns;
    for I := 1 to Count do
      List[I] := I;
    {Remove self from the list}
    List[GetConnNo] := List[Count];
    Dec(Count);
  end;
end;

```

AllConnections returns a connection list containing all valid stations except for the sender's.

If a message is to be sent to just those stations whose user has logged in with a particular name, then the following routine can be used:

```

procedure NamedConnection(Name : String; var Connect : ConnectionList);
var
  I : Word;
  Conn : Word;
  ServerInfo : ServerInformation;
begin
  GetServerInfo(ServerInfo);
  with Connect do begin
    Count := 0;
    {Add all matching connections to the list}
    for I := 1 to ServerInfo.MaxConns do begin
      Conn := GetConnFromName(I, I, Name);
      if Conn <> 0 then begin
        Inc(Count);
        List[Count] := Conn;
      end;
    end;
  end;
end;

```

After calling NamedConnection, the caller should assure that Connect.Count is not zero. Once the target connection list is initialized, it's easy to broadcast a message:

```

procedure Broadcast(Msg : BroadcastStr; var Connect : ConnectionList);
var
  I : Word;
  Result : ConnectionList;
begin
  SendBroadcastMsg(Msg, Connect, Result);
  (See who may have received it)
  for I := 1 to Result.Count do
    if Result.List[I] <> 0 then
      (Connect.List[I] wasn't connected) ;
  end;
end;

```

The longest possible broadcast message is 55 characters. The checking loop shown above is largely useless because it determines only whether a given workstation was logged on to the network, not whether it received the message. By default, the message will be displayed on line 25 of the receiver's screen. However, any workstation can call <SetBroadcastMode> to change the default; as a result, the broadcast message may be ignored altogether or stored in a buffer and retrieved only later by calling <GetBroadcastMsg>.

Piped messages start with the same connection list, but require additional steps before the message is sent. A pipe must be opened by both sender and receiver, thus requiring a degree of cooperation between the two. Each party first creates a connection list using a routine like AllConnections or NamedConnection. Then each attempts to open its half of the pipe as follows:

```

function UserBreak : Boolean;
begin
  IPXRelinquish;
  UserBreak := KeyPressed;
end;

procedure OpenPipe(Target : ConnectionList; var Actual : ConnectionList);
var
  I : Word;
  Result : ConnectionList;
begin
  repeat
    OpenMessagePipe(Target, Result);
    Actual.Count := 0;
    for I := 1 to Result.Count do
      if Result.List[I] = 0 then begin
        inc(Actual.Count);
        Actual.List[Actual.Count] := Target.List[I];
      end;
    until (Actual.Count > 0) or UserBreak;
  end;
end;

```

There is plenty of room for variation here. OpenPipe exits when any connection responds or when any key is pressed. Instead it could wait for all requested connections to respond (by specifying Actual.Count = Target.Count in the until

loop); it could try for a certain period of time and give up without user intervention; it could require a specific key to be pressed before it exits. In any case, the caller should assure that Actual.Count is greater than zero after OpenPipe returns.

After the pipe is opened, a message is sent as follows:

```
SendMessagePipe('this is the message', Actual, Result);
```

The message can be at most 126 characters long. Result can be checked as in the previous examples to determine the transmission's degree of success.

The receiver would then make the following call to get the message:

```
GetMessagePipe(SourceConn, Msg);
```

SourceConn returns the connection number of the sender whose message was received and Msg will hold the message itself. If no message is pending, SourceConn will hold zero, and Msg will be the empty string. The receiver should usually call <GetMessagePipe> repeatedly to assure that the sender's message had time to arrive. Note that each workstation can buffer up to six received messages. Subsequent calls to <GetMessagePipe> will return them in the order that they were received.

Pipes are two-way connections, so the receiver can reply to the message if appropriate.

When no more piped messages are needed, sender and receiver should close all connections, including both the fully-connected pipes and the half-opened connections:

```
Status := CloseMessagePipe(Actual, Result);  
if Status <> 0 then  
  (Error handling) ;
```

There isn't anything an application can do about a non-zero Status code, unless it is the result of a programming error (attempting to close a pipe that was never opened). Hence, the application can usually ignore an error here.

Types

```
ConnectionList =  
  record  
    Count : Byte;  
    List : array[1..255] of Byte;  
  end;
```

Used by some of the messaging routines to represent the list of network nodes to receive a message. The maximum number of connections is NetWare version-specific (e.g., NetWare ELS II supports a maximum of 8 nodes).

```
ConnInfoType =  
  record  
    ObjectID : LongInt;  
    ObjectType : Word;  
    ObjectName : String[48];  
    LoginDate : NovDateType;  
  end;
```

Describes the data structure returned by the <GetConnInfo> routine. <ObjectID> is a number used internally by NetWare, <ObjectType> equals 1 for a User or 2 for a Group, <ObjectName> is the login name, and <LoginDate> is the date and time when the login occurred.

```
NovDateType =  
  record  
    Year, Month, Day, Hour, Minute, Second : Byte;  
    WeekDay : DayOfTheWeek;  
    Filler : Byte;  
  end;
```

Record type used within <ConnInfoType>. The fields are self-explanatory.

BroadcastToConsole

Declaration

procedure BroadcastToConsole(Message : Str80);

Purpose

Send a message to the file server console.

Description

This routine will send a string of at most 55 characters. The message will be seen only if the server is acting in console mode. New messages overwrite old messages.

See Also

SendBroadcastMsg

SetBroadcastMode

CheckMessagePipe / CloseMessagePipe

Declaration

```
procedure CheckMessagePipe(var Connect, Result : ConnectionList);
```

Purpose

Checks the status of pipes to all specified workstation numbers.

Description

The list of workstations to check is passed in <Connect>. <Result> returns a list of status codes. Possible values for each entry in <Result> are:

- \$00 Success (message pipe is complete).
- \$FE Incomplete pipe (server to destination pipe doesn't exist).
- \$FF Failure (connection number invalid or sender to server pipe doesn't exist).

See Also

CloseMessagePipe

OpenMessagePipe

Declaration

```
function CloseMessagePipe(var Connect, Result : ConnectionList) : Byte;
```

Purpose

Closes pipes previously opened by <OpenMessagePipe>.

Description

<CloseMessagePipe> can close pipes with one or more connections simultaneously. The <Connect> parameter lists all the connection numbers with which a pipe is to be closed. The <Result> parameter returns the status of each attempted disconnect. For each element in the connection list, the following result codes are possible:

- \$00 Success (pipe is broken).
- \$FD Failure (target station is invalid or no longer in use).
- \$FF No such pipe exists.

When the caller closes its half of a pipe, any messages that may have been available for receipt from that pipe are lost. However, any messages previously sent by the caller through that pipe will still be available to the station at the other end.

The function returns a status code:

- \$00 Success.
- \$FC Message queue is full (pipes not closed).
- \$FF I/O error or out of dynamic workspace.

See Also

CheckMessagePipe

OpenMessagePipe

GetBroadcastMode / GetBroadcastMsg

Declaration

function GetBroadcastMode : Byte;

Purpose

Return the workstation's current broadcast mode.

Description

The broadcast mode specifies how workstation and server react to broadcast messages. It is interpreted as follows:

Mode	Server	Shell
----	-----	-----
0	Store	Retrieve and display
1	Reject	Retrieve and display
2	Reject	Ignore
3	Store	Ignore

When "Server" is "Reject", the server ignores all messages intended for the current workstation (with the exception of messages broadcast by the server itself, which are always sent). When "Shell" is "Retrieve and display", the NetWare shell immediately displays the broadcast message on line 25 of the terminal. Otherwise, the workstation may poll for messages by calling <GetBroadcastMessage>. The default broadcast mode is 0. Each workstation has its own broadcast mode.

The broadcast services are capable of storing just one message per workstation. If a broadcast message is already stored when a message is received from another workstation, that message is rejected. However, if the server sends a message to the same workstation, it immediately overwrites any pending message.

See Also

SetBroadcastMode

Declaration

procedure GetBroadcastMsg(var Message : Str80);

Purpose

Returns a broadcast message if one is available.

Description

If no broadcast message is available, <Message> returns the empty string. The length of the returned string will always be 55 or less. <GetBroadcastMsg> is useful only in broadcast modes 2 or 3; in the other modes, broadcast messages are either rejected or displayed on the screen by the NetWare shell.

See Also

GetBroadcastMode

GetConnFromName

Declaration

```
function GetConnFromName(LoConn, HiConn : Byte; ObjName : String) : Byte;
```

Purpose

Return a connection number for a log-in name.

Description

This routine searches all connected workstations ranging from <LoConn> to <HiConn> to find one with the log-in name given by <ObjName>. Searching is not case-sensitive. <GetConnFromName> returns the first connection number found with the given name. If none is found, it returns 0.

This routine is useful for determining a list of the connection numbers where broadcast or piped messages are to be sent. In combination with <GetInternetAddress>, it is also useful for determining physical addresses to be used in IPX or SPX communications.

<GetConnFromName> ignores connection numbers that fall outside the valid range for a particular version of NetWare. (ELS II, for example, supports connections ranging from 1 to only 8.) Except when dealing with duplicate log-in names, it is always safe to specify the range 1..255.

See Also

GetConnInfo

GetConnNo

Declaration

```
procedure GetConnInfo(ConnNo : Byte; var ConnInfo : ConnInfoType);
```

Purpose

Return information about a particular connection.

Description

Information returned includes the object's ID, the object type, the object's name, and the time and date it logged on to the connection. If <ConnNo> specifies an invalid connection number, <ConnInfo> is filled with nulls. See type <ConnInfoType> earlier in this section for more information.

Example

```
var
  ConnInfo : ConnInfoType;
...
  GetConnInfo(GetConnNo, ConnInfo);
  with ConnInfo, LoginDate do begin
    Writeln('ObjectID ', ObjectID);
    Writeln('Login time ', Month, '/', Day, ' at ', Hour, ':', Minute);
  end;
```

Displays information about the current workstation's connection.

See Also

GetConnNo

GetConnFromName

Declaration

```
function GetConnNo : Byte;
```

Purpose

Return the connection number of the current workstation.

Description

Each workstation on the network is given a connection number based on when it logged on to the server. Many routines, such as the message pipe services, require that the connection number be known.

Various methods are available to determine the connection number of another workstation. If the log-in name of a user on another station is known, the <GetConnFromName> routine will associate the connection number with the user name. A shared file holding the connection numbers of all active stations is another method.

See Also

GetConnFromName

ServerConnNo

GetInternetAddress / GetMessagePipe

Declaration

```
procedure GetInternetAddress(ConnNo : Byte; var InternetAdd : IPXAddress);
```

Purpose

Get Internet address for a given connection.

Description

Use this routine to translate a known connection number to a physical address suitable for sending IPX and SPX messages. <GetInternetAddress> returns valid entries for the <Network> and <Node> fields of the <IPXAddress>. The <Socket> number returned identifies the socket the local NetWare shell uses to communicate with the server. This socket must not be used by other applications.

If the specified <ConnNo> is invalid, a zero-filled Internet address is returned.

Example

See "Advanced Message Services" later in this section.

See Also

GetConnFromName

IPXInternetAddress

Declaration

```
procedure GetMessagePipe(var SourceConn : Byte; var Msg : MessageStr);
```

Description

Retrieves the message at the front of the caller's pipe message queue on the default file server. If no message is available, <Msg> is set to the empty string and <SourceConn> is set to zero. Otherwise, <Msg> contains the message, and <SourceConn> has the connection number of the sender. The message will be at most 126 characters long.

See <OpenMessagePipe> and <SendMessagePipe> for more information.

Example

See examples at the beginning of this section and in MESEXAMP.PAS.

See Also

OpenMessagePipe

SendMessagePipe

LogNetworkMessage / OpenMessagePipe

Declaration

```
procedure LogNetworkMessage(Message : Str80);
```

Purpose

Appends message to file server's NET\$LOG.MSG file.

Description

NetWare automatically prefaces the message with the current date and time, as well as the caller's connection number. This routine is intended to be used by programs that need to log usage information for accounting or other administrative purposes.

Declaration

```
procedure OpenMessagePipe(var Connect, Result : ConnectionList);
```

Purpose

Open a list of pipes.

Description

To successfully send a message by pipe, two steps must occur first. The sending station must create "half of a pipe" for each station with which it wishes to communicate. The receiving station must open its half as well, completing the connection. Both the sending and receiving halves are opened with the <OpenMessagePipe> procedure. The server acts as an intermediary for the pipe, so actually both halves are opened between a station and the server.

<OpenMessagePipe> can open pipes with one or more connections simultaneously. The <Connect> parameter lists all the connection numbers with which a pipe is requested. The <Result> parameter returns the status of each attempted connection. For each element in the connection list, the following result codes are possible:

\$00 Success (pipe is now complete and open).

\$FE Incomplete pipe (target station exists, but has not offered its half).

\$FF Failure (connection number invalid or not in use).

Warning: NetWare 386 does not support this function.

Example

See examples at the beginning of this section and in MESEXAMP.PAS.

See Also

CloseMessagePipe

SendMessagePipe

PreferredServerConnNo

Declaration

```
function PreferredServerConnNo : Byte;
```

Purpose

Returns connection ID of preferred server.

Description

This function returns zero unless a preferred server has been specified by calling <SetPreferredServerConn>. Otherwise it returns a connection ID in the range of 1..8. The connection ID is a different kind of number than a connection number such as that returned by <GetConnNo>. More information may be found with the description of <SetPreferredServerConn>.

See Also

ServerConnNo

SetPreferredServerConn

PrimaryServerConnNo / SendBroadcastMsg

Declaration

```
function PrimaryServerConnNo : Byte;
```

Purpose

Returns connection ID of primary server.

Description

The primary server is the one to which the workstation first logged on. The connection ID is a number in the range of 1..8. It is a different kind of number than a connection number such as that returned by <GetConnNo>. More information may be found with the description of <SetPreferredServerConn>.

See Also

ServerConnNo

SetPreferredServerConn

Declaration

```
procedure SendBroadcastMsg(Msg : BroadcastStr;  
                           var Connect, Result : ConnectionList);
```

Purpose

Sends a broadcast message to all connections specified in <Connect>.

Description

The message is at most 55 characters long. <Result> returns a code for each connection specified. For each element in <Connect> the following codes will be returned in <Result>:

\$00 Successful delivery.

\$FC Target station already holding message. (message rejected)

\$FD Bad connection number.

\$FF Target connection invalid. (message rejected)

See <SetBroadcastMode> for information about how the receiving workstation will react to a broadcast message.

Example

See the introduction to this section as well as MESEXAMP.PAS.

See Also

GetBroadcastMsg

SetBroadcastMode

SendMessagePipe

Declaration

```
procedure SendMessagePipe(Msg : MessageStr;  
                           var Connect, Result : ConnectionList);
```

Purpose

Sends a message to a list of workstations by using previously opened pipes.

Description

Each message may be no longer than 126 bytes. Any station may have at most 6 received messages pending at one time; additional ones are rejected.

<Connect> specifies the workstations to send to. <Result> returns the status of each delivery, using the following codes in the <List> field:

\$00 Success (message entered into pipe).

\$FC Pipe full (message rejected).

\$FE Incomplete pipe (pipe from server to destination doesn't exist).

\$FF Failure (connection number invalid or sender to server pipe doesn't exist).

Piped messages are routed through the file server and thus require server processing time. They are designed for applications requiring a moderate level of node to node communication. IPX or SPX services should be used when more substantial communication is required. NetWare 386 no longer supports piped messages.

Example

See the introduction to this section as well as MESEXAMP.PAS.

See Also

GetMessagePipe

OpenMessagePipe

Declaration

```
function ServerConnNo : Byte;
```

Purpose

Return the connection ID of the default file server.

Description

The server and workstation connections are independently numbered. In other words, the first server is server connection 1, while the first workstation logging on to it is workstation connection number 1. The file server connection ID is often referred to as the server slot number in the Novell documentation.

The default server ID is related to the primary and preferred server ID's. Here's the algorithm NetWare uses:

```
if PreferredServerConnNo <> 0 then
  ServerConnNo := PreferredServerConnNo
else if DefaultDrive = NetworkDrive then
  ServerConnNo := conn ID of server with DefaultDrive
else
  ServerConnNo := PrimaryServerConnNo;
```

See Also

PreferredServerConnNo
SetPreferredServerConn

PrimaryServerConnNo

SetBroadcastMode

Declaration

```
procedure SetBroadcastMode(Mode : Byte);
```

Purpose

Sets the broadcast mode to the value specified in <Mode>.

Description

The broadcast mode specifies how workstation and server react to broadcast messages. It is interpreted as follows:

Mode	Server	Shell
----	-----	-----
0	Store	Retrieve and display
1	Reject	Retrieve and display
2	Reject	Ignore
3	Store	Ignore

When "Server" is "Reject", the server ignores all messages intended for the current workstation (with the exception of messages broadcast by the server itself, which are always sent). When "Shell" is "Retrieve and display", the NetWare shell immediately displays the broadcast message on line 25 of the terminal. Otherwise, the workstation may poll for messages by calling <GetBroadcastMessage>.

If you modify the broadcast mode, it is advisable to save the current state of the broadcast mode prior to changing it and then to restore it to the original value before terminating the program.

Declaration

```
procedure SetPreferredServerConn(ServConnNo : Byte);
```

Purpose

Specify a preferred server.

Description

This routine is useful for NetWare installations that have multiple file servers, where a given workstation may deal with more than one server at a time. Novell defines some terminology for discussing multiple servers, and we need to describe that first.

The "primary" file server is the server to which a station first logs on. The "default" server is the one where the user's current drive:directory is located, unless the default drive is local to the workstation. The default server will always be the same as the primary server unless the station has attached to additional servers. Finally, the "preferred" server is one that is defined specifically by a given application. There is no preferred server unless the application asks for one; even that value is reset when the application ends.

For most communications between a workstation and a file server, a drive letter uniquely identifies which server is intended. For workstation requests that cannot be referenced through a drive letter, the NetWare shell routes the request using the following priorities:

1. preferred server
2. default server
3. primary server

Hence, the <SetPreferredServerConn> procedure may be used to specify the server to which such requests are routed. <ServerConnNo> is always a number in the range 0..8, where zero means that no preferred server is active.

Example

```
ServerConn := PreferredServerConnNo;  
SetPreferredServerConn(2);  
...  
SetPreferredServerConn(ServerConn);
```

Temporarily changes the preferred server to number 2 (the second one that the workstation logged on to), performs some actions, then restores the original preferred server.

See Also

PreferredServerConnNo
ServerConnNo

PrimaryServerConnNo

Advanced Message Services

NetWare supports two advanced methods of communicating between workstations. These methods offer higher performance and larger data capacity than the basic pipe and broadcast services, and they don't use any server resources at all. These advanced methods are referred to as IPX and SPX services.

IPX stands for Internetwork Packet eXchange protocol. It is Novell's implementation of the Internetwork Datagram Protocol designed by Xerox. The term "internetwork" is used to refer to the current network as well as any other networks that may be bridged to it. IPX requires Advanced NetWare version 2.0 or later. Applications running under Advanced NetWare use the IPX drivers to communicate directly with other workstations, servers, or devices.

Each device on the internetwork is called a node and has a unique address. IPX enables a node to send a packet to, or receive a packet from, any other node. Packets are arbitrary byte sequences as long as 546 bytes. IPX isolates the application from the details of the packet's physical routing. Although IPX does not guarantee delivery of a packet (the other node might still ignore it), guaranteed delivery protocols can be built using its services. SPX (described in the following) is a guaranteed delivery protocol based on IPX.

The NETWARE unit implements IPX routines ranging from low to high levels of integration. The low level routines are simply Pascal procedures and functions that call the NetWare IPX services directly. The higher level routines isolate the programmer from the details of forming IPX packets. Application developers should use the high level routines. Because the low level routines require such detailed understanding of the IPX data structures, we haven't documented them here. The interested programmer is directed to the Novell API Reference Volume 2.

Two important data types associated with IPX services are the IPX header, `<IPXHeader>`, and the event control block, `<IPXECB>`. `<IPXHeader>` contains routing information used by IPX to deliver a packet. `<IPXECB>` includes a pointer to an `<IPXHeader>` and additional pointers to one or more data buffers that store the packet itself.

In order to send a packet to another node, several fields in the `<IPXHeader>` must be specified:

- the network address, a four byte number
- the physical node address, a six byte number
- the socket, a two byte number

The network address specifies the *network* of the destination node (required since IPX will bridge networks). Storing zero in this field indicates the same network as that of the sender.

The node address specifies a particular station within the network. The procedure <IPXInternetAddress> is supplied to return the network and node address of the calling station, and <GetInternetAddress> will return the address of a given connection on the current network.

In addition to the node and network addresses, sending and receiving workstations must agree on a "socket". You can think of sockets as two-way radio channels; both sender and receiver must be tuned to the same channel for communications to occur. A node may have up to 20 sockets open at a time. (In Advanced NetWare versions 2.11 and later you can specify the socket capacity of a workstation. See Shell Configuration File Options in the NetWare Supervisor Reference)

The caller must specify a socket number that doesn't conflict with other sockets currently active on the network. Socket numbers ranging from \$4000 to \$7FFF are called "dynamic" sockets; these are fair game for any user of the network. Values \$8000 and larger are "well-known" sockets, whose values have been reserved by Novell for approved applications. Sockets less than \$4000 are reserved by Novell for internal use.

An application can use any of several approaches to select unique but agreed-upon socket numbers. The first is easy and generally quite adequate: simply use an arbitrary number from the range of dynamic sockets and make this choice an installation option for the application. In the unlikely event that there is a conflict with some other program running on the network, the network administrator can choose a different number and reinstall the program.

The second is more complicated, but works without human intervention. When the application starts up, open a pipe to all other nodes (more on this in the basic messaging section). Then send a message over this pipe and listen for a response. If one is received, assure that it starts with an application-specific recognition code, and interpret the rest of the message as one or more socket numbers to use for IPX communications. If no valid response is received, then call a supplied NETWARE routine to get a unique socket number. During the remaining time the application runs, monitor the still-open pipe to respond to a request for the common socket number from another station.

A similar approach uses a shared file. The file contains two or more word values: first, a count of the participating workstations; and then one or more socket numbers. When the first station starts the program, it checks for the existence of the

file. If it doesn't exist, or if the station count is zero, it allocates a unique socket and writes out a count of 1 followed by the socket number. Otherwise, it increments the station count and uses the socket number stored there. When the application halts, it decrements the station count. Of course, proper file locking is necessary to avoid timing problems.

To summarize this brief discussion of IPX services, we include the following list of functions provided by the NETWARE unit:

- determine whether IPX services are available
- get workstation network address
- open, close, or allocate a socket
- send or listen for a message
- cancel send or listen request
- relinquish control to IPX driver

Advanced NetWare 2.0 implements a higher level of node-to-node communications called SPX (Sequenced Packet Exchange Protocol). SPX is based on the Sequenced Packet Protocol (SPP) defined by Xerox.

Building upon the IPX services, SPX provides a guaranteed delivery system for packets of data. Packets that are not acknowledged by the recipient within a specified time are automatically retransmitted. After successive retransmissions have failed, the connection is assumed to be broken. SPX automatically selects appropriate timeout and retransmission counts based on the physical characteristics of the network hardware. Hence, applications using SPX need not be concerned with the details of the guaranteed delivery system. Because SPX has more overhead than IPX, the maximum packet size is slightly smaller: 534 bytes.

Unlike pipes and broadcast messages, an SPX connection transfers messages between just two nodes at a time. During the establishment of an SPX connection, one node defines itself as the sender and the other node as the receiver. The initial connection is *one-way*; the receiver cannot reply to messages. Although it is possible to make the connection two-way, it isn't easy to encapsulate the method in a general purpose routine. Hence, for two-way SPX connections, you have two alternatives. You can open a separate connection for the reverse direction; this works fine, but costs a bit in extra memory and processor overhead. Or you can take a look at the supplied demonstration programs NSEND and NRECEIVE, which show how to reverse the connection.

Calls to Novell's IPX and SPX low-level routines do nothing more than submit a request for a particular message event. The actual completion of the event does not occur until an undetermined amount of time later. NetWare requires a separate event control block (of type <IPXECB>) to manage each pending event. Establishing an SPX connection between two stations actually requires two event control blocks on the sender's side--one for sending out the request, and another to receive confirmation from the receiver.

The receiver of SPX messages may require that even more event control blocks be available simultaneously. For example, suppose the receiver expects to receive 10 packets of information. The receiver could declare just one event control block. It would listen for each packet, process it upon arrival, and quickly make the single <IPXECB> available for listening again. However, this would leave open a window when an incoming packet might not find an <IPXECB> available, resulting in a lost packet. A more reliable approach would be to make two event control blocks available, in order to guarantee that one is always ready while the other is temporarily tied up. In some cases, it might even be desirable to make 10 event control blocks available; in this way the entire transaction could be completed without resubmitting control blocks to SPX.

The SPX routines in the NETWARE unit make it easy to deal with these situations by automatically managing a pool of event control blocks. When an application establishes an SPX connection, it simply specifies the number of event control blocks and the maximum size of each data packet. The demonstration program NRECEIVE is a good example of the use of these pools and their associated procedures and functions.

The NETWARE unit supports SPX services at both low and high levels. Again, we've focused on documenting the higher level routines, since the others require more background information than we can provide in a reasonable space.

The NETWARE unit provides the following facilities for accessing SPX:

- determine whether SPX services are available
- establish or terminate an SPX connection
- send or listen for a message
- cancel send or listen request

Examples

The supplied program MESEXAMP.PAS demonstrates a number of the methods used to pass messages on NetWare and NetBIOS LANs. This section will introduce the NetWare IPX and SPX methods used in that program. Before using either method, the program must assure that NetWare, IPX, and SPX are available:

```
procedure IPXSPXAvail(var IPXOK, SPXOK : Boolean);
var
  Logged : Boolean;
  Version : Word;
  MaxConn : Word;
  AvailConn : Word;
begin
  IPXOK := False;
  SPXOK := False;
  if NetWareLoaded(Logged) then
    if Logged then begin
      IPXOK := IPXServicesAvail;
      if IPXOK then
        SPXOK := SPXServicesAvail(Version, MaxConn, AvailConn);
    end;
  end;
end;
```

The next task of any application using IPX or SPX is to determine the Internet address of the receiver. When using IPX, a message may be sent to all nodes using the special physical address <AllNodes>. The procedure <GetInternetAddress> may be used for either IPX or SPX:

```
var
  Receiver : IPXAddress;

  GetInternetAddress(ReceiverConn, Receiver);
```

ReceiverConn is a NetWare connection number (1..100, or 1..255 under NetWare 386). It can be determined from a login name by using <GetConnFromName> as shown in the examples for broadcast and pipe messages.

The sender and receiver must also agree on a socket number. As discussed previously, the usual method is to pick an arbitrary number that can be modified by the network administrator:

```
const
  IPXSocket : Word = $7001;
```

The following function shows how to send an IPX message, using the global socket number.

```

function SendIPX(Receiver : IPXAddress;
                 MsgLen : Word;
                 var Msg) : Byte;
var
    Status : Byte;
    IEvent : IPXRec;
begin
    {Send the message}
    Status := IPXSend(IEvent, Receiver, IPXSocket, False, MsgLen, Msg);
    if Status <> 0 then begin
        {MsgLen too large, or socket table full}
        SendIPX := Status;
        Exit;
    end;
    {Wait for the message to depart}
    while not IPXEventComplete(IEvent, Status) do ;
        {Status will be zero for success}
        SendIPX := Status;
    end;
end;

```

The message may be up to 546 bytes long. The example uses an untyped parameter to allow any data type to be sent. <IPXSend> opens the specified socket if it isn't already open. SendIPX returns a status code indicating whether the message could be sent. If a non-zero code is returned, the caller can handle the error accordingly.

The receiver would use the following function, again referring to the global socket number.

```

function ReceiveIPX(MaxLen : Word;
                   var Msg) : Byte;
var
    Status : Byte;
    IEvent : IPXRec;
begin
    {Listen for the message}
    Status := IPXListen(IEvent, IPXSocket, False, MaxLen, Msg);
    if Status <> 0 then begin
        {MaxLen too large, or socket table full}
        ReceiveIPX := Status;
        Exit;
    end;
    {Wait for a message to arrive}
    while not IPXEventComplete(IEvent, Status) do
        if KeyPressed then begin
            {User got impatient}
            ReceiveIPX := 2;
            Exit;
        end;
        {Status will be zero for success}
        ReceiveIPX := Status;
    end;
end;

```

Note that ReceiveIPX does not return the actual length of the message. Instead the caller must specify the maximum length that the Msg variable can hold. IPX will

return an error code if the actual message is longer than MaxLen. The caller can then call ReceiveIPX again to get the rest of the message. If message length varies, it is the application's responsibility to encode the length within the message itself.

It is important for ReceiveIPX to incorporate some kind of timeout mechanism, in case a message never arrives. Here the listen cycle is broken when the user presses a key; an alternate approach would be to wait a certain amount of time before exiting with an error code.

Before the application halts, both sender and receiver must close the socket with the following call:

```
IPXCloseSocket(IPXSocket);
```

SPX messages are more reliable, but more complicated to use. As with pipes, the sender and receiver must establish a connection prior to exchanging messages. The sending program establishes its side of the connection with the following sequence:

```
const
    SPXSocket : Word = $7000;

var
    SEvent : SPXRec;

function SPXSendConn(Receiver : IPXAddress) : Byte;
var
    Status : Byte;
begin
    Status := SPXEstablishConn(SEvent, Receiver, SPXSocket, False, 0, 2, Nil);
    if Status <> 0 then begin
        {Connection or socket table full}
        SPXSendConn := Status;
        Exit;
    end;
    {Wait for acknowledgement of connection}
    while not SPXEventComplete(SEvent, Status) do
        if KeyPressed then begin
            {User got impatient}
            SPXSendConn := 2;
            Exit;
        end;
    {Status will be zero for success}
    SPXSendConn := Status;
end;
```

Note that the SEvent variable is stored outside of SPXSendConn. This is important, because the routine that sends the messages must refer to the same variable. As in the other examples, SPXSendConn uses a simple approach to detect a failed connection: it tries continuously until either it succeeds, SPX times out, or the user presses a key. SPX will eventually time out and return a corresponding error code,

but this may take longer than the user is willing to wait. SPXSendConn sets up an SPX connection where the sender can send but not receive messages.

The receiver establishes its side of the connection with the following routine:

```
const
    SPXSocket : Word = $7000;

var
    SEvent : SPXRec;

function SPXReceiveConn(MaxLen : Word) : Byte;
var
    Status : Byte;
begin
    Status := SPXListenForConn(SEvent, SPXSocket, False, MaxLen, 2, Nil);
    if Status <> 0 then begin
        {Connection or socket table full}
        SPXReceiveConn := Status;
        Exit;
    end;
    {Wait for acknowledgement of connection}
    while not SPXEventComplete(SEvent, Status) do
        if KeyPressed then begin
            {User got impatient}
            SPXReceiveConn := 2;
            Exit;
        end;
        {Status will be zero for success}
        SPXReceiveConn := Status;
    end;
```

The call to <SPXListenForConn> allocates two buffers of size MaxLen on the heap, as well as the event control blocks needed to listen for incoming messages. The largest possible SPX message is 534 bytes. Otherwise, this routine works much like SPXSendConn.

To allow for imperfect synchronization between sender and receiver, it may be necessary to call both SPXSendConn and SPXListenConn within loops. Once the connection has been established, the sender sends a message like so:

```
function SendSPX(DataType : Byte;
                MsgLen : Word;
                var Msg) : Byte;

var
    Status : Byte;
```

```

begin
  SPXSend(SEvent, False, DataType, MsgLen, Msg);
  {Wait for acknowledgement of the message}
  while not SPXEventComplete(SEvent, Status) do
    if KeyPressed then begin
      {User got impatient}
      SendSPX := 2;
      Exit;
    end;
    {Status will be zero for success}
    SendSPX := Status;
  end;
end;

```

Although this looks much like the corresponding IPX routine, there is an important difference. When the IPX routine returns a zero result, it means that the message was successfully sent. A zero result from the SPX routine means that the message was successfully *received*. This is the crux of the SPX guaranteed delivery feature.

Also note that the caller can specify a *DataType* for the message. This byte provides a convenience to the receiver, allowing it to classify messages of differing types without looking into each message.

The receiver's code is quite a bit different from what we've shown so far.

```

function ReceiveSPX(var DataType : Byte;
                   MaxLen : Word;
                   var Msg) : Byte;
var
  Status : Byte;
  PoolIndex : Byte;
  P : Pointer;
begin
  {Wait for receipt of the message}
  while not SPXListenPooled(SEvent, Status, PoolIndex, DataType, P) do
    if KeyPressed then begin
      {User got impatient}
      ReceiveSPX := 2;
      Exit;
    end;
    if Status = 0 then
      {Message received, return it to caller}
      Move(P^, Msg, MaxLen);
      {Resubmit the event control block for listening}
      SPXReplenishPool(SEvent, PoolIndex);
      {Status will be zero for success}
      ReceiveSPX := Status;
    end;
  end;
end;

```

ReceiveSPX waits for an incoming message or until a key is pressed. If a message is received, SPX automatically acknowledges it. The NETWARE unit buffers the message temporarily on the heap, and ReceiveSPX copies that to the program's

variable. It then resubmits the event control block used to receive the message to the pool of blocks available for listening.

Before the application halts, it is important for both sender and receiver to break the connection. Each can make the following call:

```
SPXTerminateConn(SEvent);
```

Options such as interrupt-driven messaging and two-way connections are also available to the SPX programmer. See the demonstration programs and the remainder of this section for more details.

Constants

```
AllNodes : PhysicalNodeAddress = ($FF,$FF,$FF,$FF,$FF,$FF);
```

When passed as a node address to an IPX send routine, this value indicates that the message should be sent to all nodes. Novell warns that not all hardware configurations support this special address. Even when it is supported, only nodes that are listening through the specified socket will actually receive the message. A destination address of <AllNodes> is not allowed for SPX messages.

```
MaxECBPool = 32;
```

Maximum number of event control blocks for a given SPX connection.

```
PoolSocketLongevity : Boolean = False;
```

Controls how long sockets used by the high level SPX routines remain open. When True, sockets remain open until explicitly closed; when False, they are closed automatically when the program terminates. Should be set True only for memory resident programs that process messages after going resident.

```
SPXRetryCount : Byte = 0;
```

Specifies the retry count for SPX operations. A value of zero indicates that SPX should use its predetermined count; any other value is used explicitly.

Types

Many of the types described in this section need not be accessed directly by an application. Descriptions are included to illustrate the hierarchy of information leading to the high level types passed as parameters to the NETWARE routines.

Although the data types refer to standard Pascal identifiers, you should know that NetWare reverses the byte ordering of almost all Word and LongInt fields. That is, the most significant byte is stored first instead of last. The high level routines in the NETWARE unit reverse the order as necessary, for example when storing a socket number you've passed as a parameter.

```

FragmentDescriptor =
  record
    Address : Pointer;
    Size : Word;
  end;

```

Holds the address and size of a "fragment". A packet is a collection of fragments, which are often used to ease the receiver's job of separating the packet's components.

```

IPXAddress =
  record
    Network : LongInt;
    Node : PhysicalNodeAddress;
    Socket : Word;
  end;

```

Represents the three components of an IPX internetwork address: the network number, the physical node address, and the socket number.

```

IPXECB =
  record
    Link : Pointer;
    ESRAddress : Pointer;
    InUse : Byte;
    CompletionCode : Byte;
    SocketNumber : Word;
    IPXWorkspace : LongInt;
    DriverWorkspace : array[1..12] of Byte;
    ImmediateAddress : PhysicalNodeAddress;
    FragmentCount : Word;
    FD1 : FragmentDescriptor;
    FD2 : FragmentDescriptor;
    FD3 : FragmentDescriptor;
    FD4 : FragmentDescriptor;
  end;

```

The control block for an IPX event; an event is the process of sending or listening for a message. Although the NETWARE unit manages the contents of this record for you, a little description may clarify how it works.

<ESRAddress> may hold the address of an "event service routine". If the pointer isn't Nil, NetWare will call the routine it points to when an event is completed. This forms the basis for interrupt driven messaging protocols. Further information regarding event service routines is provided with the <SPXListenForConn> routine. This field is typically initialized to Nil.

The <InUse> field is initialized to zero before submitting an event control block to IPX. IPX sets the field to a non-zero value to specify the event's status. When the field is zero again, the event is complete.

Once an event is complete, <CompletionCode> specifies its outcome. See the <IPXEventComplete> function for a list of the possible values.

<SocketNumber> is the socket being used for communication.

<ImmediateAddress> holds the address of a node within the current network. If the message originated from, or is destined for, another network,

<ImmediateAddress> holds the address of the internetwork bridge node.

A data packet may be composed of several fragments. The first fragment must always begin with an <IPXHeader>. Remaining fragments may hold the actual packet data. The NETWARE unit always uses two fragments, one for the header and the other for the data. The sum of the <Size> fields of all the individual fragments must not exceed 576 bytes.

```
IPXHeader =  
  record  
    CheckSum : Word;  
    Len : Word;  
    TransportControl : Byte;  
    PacketType : Byte;  
    Destination : IPXAddress;  
    Source : IPXAddress;  
  end;
```

Used by IPX to manage a message. <Checksum> is used internally for error control. <Len> is the length of the transmitted packet, including both the header and the data. Valid lengths range from 30 to 576. <TransportControl> is used internally. <PacketType> must be either 0 (unknown type) or 4 (packet exchange packet) when used with IPX. <Destination> and <Source> specify the internetwork addresses of the receiver and sender, respectively. <IPXHeader> variables are managed for you by the NETWARE unit, so you will use them only for information, if at all.

```
IPXRec =  
  record  
    IPXHead : IPXHeader;  
    ECB : IPXECB;  
  end;
```

The high level type used by NETWARE to manage IPX messages. A variable of this type must be declared by the calling program and passed to the various IPX message routines. The <IPXHead> field describes the source and destination of the message. The <ECB> field describes the message itself with pointers to its contents and information about message routing and status. NETWARE routines initialize this record and return information from fields internal to it.

```
PhysicalNodeAddress = array[1..6] of Byte;
```

Represents a physical node address, which is one part of the complete internetwork address. The physical node address appears on a sticker affixed to most Ethernet cards.

```

SPXHeader =
  record
    IPXPacket : IPXHeader;
    ConnectControl : Byte;
    DataStreamType : Byte;
    SourceConnID : Word;
    DestConnID : Word;
    Sequence : Word;
    Acknowledge : Word;
    Allocation : Word;
  end;

```

Used by SPX to manage a message. <IPXPacket> holds information passed on to the lower level IPX services. With one exception, the remaining fields are for internal use by SPX. The <DataStreamType> field holds a user-defined value that specifies the packet type (perhaps to make classification of a given packet easier for the receiver). Any values but \$FE and \$FF are acceptable. The NETWARE unit hides the details of this type from you, but does allow you to specify the <DataStreamType> in the <SPXSend> routine.

```

SPXRec =
  record
    ConnID : Word;
    SPXHead : SPXHeader;
    ECB : IPXECB;
    PoolRec : SPXPoolRec;
  end;

```

The high level type used by NETWARE to manage SPX messages. A variable of this type must be declared by the calling program and passed to the various SPX message routines. The <ConnID> field contains a "connection number", which is SPX's higher level form of an IPX socket. The <SPXHead> field describes the source and destination of the message. The <ECB> field is the main event control block, used during establishment of the connection. The <PoolRec> field hides a complex data structure; this structure manages a variable number of event control blocks (allocated on the heap), which are used while listening for messages. See the NETWARE source code for more details about the <SPXPoolRec> type. One important detail is that each of the event control blocks is referenced with an index in the range from 1 to <NumListenPoolECBs> (a parameter specified when the connection is requested). This "pool index number" is returned whenever a message is received, and must be passed to the procedure <SPXReplenishPool> to enable the associated control block for listening again.

Generally, you don't need to be concerned about the contents of an <SPXRec>. NETWARE routines initialize the record for you and return information from it as needed. You can think of it as a "file variable" for SPX.

IPXAssignUniqueSocket / IPXCancelEvent

Declaration

```
function IPXAssignUniqueSocket(var SocketNum : Word;  
                               Forever : Boolean) : Byte;
```

Purpose

Returns a valid, opened socket that is not being used by any other station on the network.

Description

This socket can then be used for sending IPX or SPX messages. It is the application's responsibility to communicate this socket number to cooperating workstations. <SocketNum> need not be initialized on entry to this routine. See <IPXOpenSocket> for a description of the <Forever> parameter.

A status code is returned in function result:

```
$00 Success.  
$FE Socket table is full. (station has 20 sockets open already)
```

See Also

IPXCloseSocket IPXOpenSocket

Declaration

```
function IPXCancelEvent(var IPXEvent : IPXRec) : Byte;
```

Purpose

Cancels any pending IPX event, including send and listen events.

Description

After the event is successfully cancelled, <IPXEventComplete> will return True and a <CompletionCode> of \$FC. The application may then reuse the event variable <IPXEvent>.

A status code is returned in function result:

```
$00 Success.  
$FF Event was not active.  
$F9 Event was active but couldn't cancel it.
```

Error \$F9 occurs if the IPX driver is in the process of transferring data to or from the fragment buffers of the packet when the cancellation request occurs. When cancelling a send event, there is no guarantee that the intended receiver won't receive the packet anyway.

See Also

IPXEventComplete

IPXCloseSocket

Declaration

procedure IPXCloseSocket(SocketNum : Word);

Purpose

Closes the specified socket.

Description

This automatically cancels any pending messages that are associated with the socket. Any further incoming packets are ignored. Attempting to close a socket that is not open has no effect.

Each workstation may have no more than 20 sockets open at a time (the limit is configurable via SHELL.CFG, however). For some applications, this may mean that it's a good idea to close each socket as soon as its event is complete.

Transient (non-TSR) applications must close all their sockets before returning to DOS. Even if the sockets were opened with the <Forever> flag set to False, there exists a small window of time after the application has halted and before the IPX manager can close the socket when an incoming packet might cause a system crash. It is best for a Turbo Pascal program to define an exit procedure that will close all opened sockets no matter how the program terminates.

If you are using event service routines, note that <IPXCloseSocket> may *not* be called within the service routine. <IPXCloseSocket> is not reentrant, and generally should not be called from within any kind of interrupt service routine.

See Also

IPXOpenSocket

IPXEventComplete / IPXInternetAddress

Declaration

```
function IPXEventComplete(var IPXEvent : IPXRec;  
                           var CompletionCode : Byte) : Boolean;
```

Purpose

Returns the status of a pending IPX send or listen event.

Description

<IPXEvent> is the event to check. When <IPXEventComplete> returns True, the event has been completed. In this case, <CompletionCode> returns the status of the event.

Call <IPXEventComplete> after submitting an IPX event with either <IPXSend> or <IPXListen>. If either one has been called with the <WaitForCompletion> flag set to True, then <IPXEventComplete> is guaranteed to return True on the first call. Otherwise, the program should poll <IPXEventComplete> until it returns True. In the meantime, the application may continue to do other work. The application should call <IPXRelinquish> regularly within the polling loop to assure that incoming packets are detected.

<CompletionCode> may return the following values:

- \$00 Packet was sent or received successfully.
- \$FC Send or listen request was cancelled.
- \$FD Malformed packet. (too small, too large, incorrect fragment size or count)
- \$FE Packet is undeliverable. (destination cannot be detected, no internetwork bridge is available, or there has been a hardware failure)
- \$FF Unable to send packet. (there has been a hardware or network failure)

See Also

IPXListen
IPXRelinquish

IPXSend

Declaration

```
procedure IPXInternetAddress(var Address : IPXAddress);
```

Purpose

Returns the internetwork address of the calling workstation.

Description

The third field of the <IPXAddress> type, the <Socket>, is not initialized by this call. Network and physical node addresses of the caller are returned.

See Also

GetInternetAddress

IPXListen

Declaration

```
function IPXListen(var IPXEvent : IPXRec; Socket : Word;  
                  WaitForCompletion : Boolean; MaxPacketSize : Word;  
                  var DataPacket) : Byte;
```

Purpose

Initializes IPX data structures to receive a packet, and submits the event to IPX.

Description

<IPXEvent> is initialized by <IPXListen> and maintains all the information required for managing the message. The caller must specify a <Socket> number for the transmission. Sender and receiver must agree on a socket number. See <IPXOpenSocket> for details. <IPXListen> will open the socket if it isn't already open.

When <WaitForCompletion> is False, <IPXListen> returns immediately after submitting the request to listen. In this case, the calling application may continue with other tasks, but it must poll the <IPXEventComplete> function to determine when the event has been completed. If <WaitForCompletion> is True, <IPXListen> itself polls until the event is complete. The socket is not automatically closed upon completion of the listen event. The completion code returned by <IPXEventComplete> will tell whether a message was actually received.

<MaxPacketSize> specifies the maximum number of bytes to receive, not including the header information managed internally by <IPXListen>. The largest allowable packet size is 546 bytes. <DataPacket> may be a variable of any type, as long as it is large enough to hold at least the number of bytes specified by <MaxPacketSize>.

Both <IPXEvent> and <DataPacket> must be variables that are static throughout the duration of the event. If they are global variables (or if <WaitForCompletion> is True), this requirement is met automatically. If they are allocated on the heap, they must remain allocated until the event is complete. If they are local variables, the routine where they are declared must not exit until the event is complete. (It may call other routines, however.)

A status code is returned in function result:

```
$00 Success.  
$01 MaxPacketSize too large.  
$FE Socket table full.
```

Example

See the examples at the start of this section, as well as MESEXAMP.PAS.

See Also

IPXCloseSocket
IPXOpenSocket

IPXEventComplete
IPXSend

Declaration

```
function IPXOpenSocket(SocketNum : Word; Forever : Boolean) : Byte;
```

Purpose

Opens a socket for use by IPX or SPX.

Description

Sender and receiver must agree on a socket number in order to communicate. Sockets numbered from \$4000 to \$7FFF are available for use by any application. Any workstation may have up to 20 sockets open at one time (configurable via SHELL.CFG).

When <Forever> is True, the socket remains open until explicitly closed, even after the current program terminates. Otherwise, NetWare will close it automatically when the program ends. Setting <Forever> True would be useful for a TSR that requires IPX services.

The high level <IPXSend> routine will open the requested socket at the time a message is sent, if necessary. Thus <IPXOpenSocket> may be most useful for determining that a socket *can* be opened before an application proceeds too far.

Even though NetWare automatically closes sockets when <Forever> is set False, it is still recommended that the application close its own sockets. See <IPXCloseSocket> for more information.

A status code is returned in function result:

```
$00 Success.  
$FE Socket table is full. (station has 50 sockets open already)  
$FF Socket already open.
```

Example

See examples near the beginning of this section as well as MESEXAMP.PAS.

See Also

IPXAssignUniqueSocket

IPXCloseSocket

IPXRelinquish

Declaration

procedure IPXRelinquish;

Purpose

Temporarily relinquish application control to NetWare.

Description

This procedure serves one of two purposes, depending on whether it is invoked from a file server or a workstation. On a server, this function temporarily suspends the calling process so that the server program gets CPU resources immediately. Similarly, on a workstation, the NetWare shell gets temporary control during which it may check for incoming and outgoing messages.

This procedure should be called frequently while an application waits for an IPX or SPX event to complete. Typically, the call will take the following form:

```
repeat
  {application does some work}
  ...
  IPXRelinquish;
  {application does more work}
  ...
until IPXEventComplete(IPXEvent, CompletionCode);
```

<IPXEventComplete> itself calls <IPXRelinquish>. Hence, a call to <IPXRelinquish> is needed only if the application is performing many tasks between calls.

See Also

IPXEventComplete

Declaration

```
function IPXSend(var IPXEvent : IPXRec; Receiver : IPXAddress;  
    Socket : Word; WaitForCompletion : Boolean;  
    DataPacketSize : Word; var DataPacket) : Byte;
```

Purpose

Initializes IPX data structures to send a packet, and submits the event to IPX.

Description

<IPXEvent> is initialized by <IPXSend> and maintains all the information required for managing the message. The caller must initialize <Receiver> to specify where the packet should go. See the routine <GetInternetAddress> for one method to determine <Receiver>'s address. The caller must also specify a <Socket> number for the transmission. Sender and receiver must agree on a socket number. See <IPXOpenSocket> for details. <IPXSend> will open the socket if it isn't already open.

When <WaitForCompletion> is False, <IPXSend> returns immediately after submitting the request to send. In this case, the calling application may continue with other tasks, but it must poll the <IPXEventComplete> function to determine when the event has been completed. If <WaitForCompletion> is True, <IPXSend> itself polls until the event is complete. The socket is not automatically closed upon completion of the send event. See <IPXCloseSocket>. Call <IPXEventComplete> to determine whether the send event was successful or not.

<DataPacketSize> specifies the number of bytes to send, not including the header information managed internally by <IPXSend>. The largest allowable packet size is 546 bytes. <DataPacket> may be a variable of any type, as long as it holds at least the number of bytes specified by <DataPacketSize>.

Both <IPXEvent> and <DataPacket> must be variables that are static throughout the duration of the event. If they are global variables (or if <WaitForCompletion> is True), this requirement is met automatically. If they are allocated on the heap, they must remain allocated until the event is complete. If they are local variables, the routine where they are declared must not exit until the event is complete. (It may call other routines, however.) If these requirements are not met, the IPX driver will overwrite memory that it shouldn't, probably leading to a program crash.

A status code is returned in function result:

```
$00 Success.  
$01 <DataPacketSize> too large.  
$FE Socket table full.
```

See Also

IPXCloseSocket
IPXListen

IPXEventComplete
IPXOpenSocket

IPXServicesAvail / SPXAbortConn

Declaration

```
function IPXServicesAvail : Boolean;
```

Purpose

Determines whether IPX services are available.

Description

IPX services are used for efficient low-level message passing from node to node.

See the introduction to this section for more information about IPX. IPX services may be available even if the NetWare shell is not loaded or the user is not logged in.

Example

```
if IPXServicesAvail then begin
  if IPXOpenSocket($7001, False) = 0 then begin
    {Perform IPX communications}
  end else
    {Bad socket} ;
end else
  {No IPX driver loaded} ;
```

Tests for IPX services before using them.

See Also

SPXServicesAvail

Declaration

```
procedure SPXAbortConn(var SPXEvent : SPXRec);
```

Purpose

Abruptly terminates an existing SPX connection.

Description

<SPXAbortConn> does not notify the connection partner of the decision to break the connection. The partner will discover that the connection is no longer valid when it sends a packet or attempts to terminate the connection itself.

This routine is designed to unilaterally break a connection when a serious error condition is encountered. Under most circumstances, <SPXTerminateConn> should be called instead.

See Also

SPXEstablishConn

SPXTerminateConn

Declaration

```
function SPXEstablishConn(var SPXEvent : SPXRec;  
    Receiver : IPXAddress;  
    Socket : Word;  
    WaitForCompletion : Boolean;  
    MaxListenDataSize : Word;  
    NumListenPoolECBs : Byte;  
    PoolESR : Pointer) : Byte;
```

Purpose

Establishes an SPX connection with a listening partner.

Description

An initial message is sent to the partner's internetwork address in an attempt to create an SPX connection. If the partner responds correctly, then a communication channel is established. The partner must call <SPXListenForConn> in order to establish its side of the link.

<SPXEstablishConn> uses the <SPXEvent> parameter as a work area, similar to the way that file operations use a file variable. You don't need to initialize this parameter prior to calling <SPXEstablishConn>. However, remember that <SPXEvent> must be continuously accessible as long as the SPX connection is active. It's safest to make the variable passed in the <SPXEvent> parameter a global variable.

The caller must initialize <Receiver> to specify the partner's address. See the routine <GetInternetAddress> for one method to determine this address. The caller must also specify a <Socket> number for the transmission. Sender and receiver should agree on a socket number. See <IPXOpenSocket> for information regarding socket numbers. <SPXEstablishConn> will open the socket if it isn't already open.

When <WaitForCompletion> is set to False, <SPXEstablishConn> returns immediately after submitting the request to establish a connection. In this case, the calling application may continue with other tasks, but it must poll the <SPXEventComplete> function to determine when the event has been completed. If <WaitForCompletion> is True, <SPXEstablishConn> itself polls until the event is complete.

<MaxListenDataSize> specifies the size of the largest data packet to be received on a two-way SPX connection. By default, the station calling <SPXEstablishConn> can only *send* messages. In this case, the correct value to specify for <MaxListenDataSize> is zero, since the only messages received will be zero-size acknowledgement packets. If the method demonstrated by NSEND and NRECEIVE is used to establish a two-way connection, then <MaxListenDataSize> must contain the size of the largest application packet.

SPXEstablishConn

<NumListenPoolECBs> must always be at least two. This allows for an event control block to listen for the connection acknowledgement, plus one spare block as recommended by Novell. Unless the sender is also a listener, <NumListenPoolECBs> need not be larger than two.

<PoolESR> is a pointer to an optional user-supplied event service routine. Passing a value of Nil deactivates this feature. Specifying a <PoolESR> would be desirable only when a high-performance, real-time, two-way SPX connection is required. See <SPXListenForConn> for more details.

The SPX "connection ID" eventually established by this function is stored in the <SPXEvent> variable.

A status code is returned in function result:

```
$00 Success.  
$EF Local connection table is full.  
$FD Malformed packet. (NETWARE internal error)  
$FE The socket table is full.
```

Because SPX implicitly handles time-outs, do not call <IPXCancelEvent> to terminate an <SPXEstablishConn> call that is in progress. Instead, call <SPXTerminateConn>.

Example

See the introduction to this section as well as MESEXAMP.PAS.

See Also

SPXEventComplete
SPXTerminateConn

SPXListenForConn
SPXSend

Declaration

```
function SPXEventComplete(var SPXEvent : SPXRec;  
                           var CompletionCode : Byte;) : Boolean;
```

Purpose

Polls the status of an SPX event to determine when the event is complete.

Description

This routine is necessary because SPX messaging services are asynchronous. When an application calls any of the following routines and the <WaitForCompletion> parameter is False, SPX submits a message header to a queue and returns immediately:

SPXEstablishConn SPXListenForConn SPXSend

Only later, when the message has made its way across the network, is the event considered complete. At that time, the caller can check the outcome, e.g., whether the connection was established or the message was successfully sent.

<SPXEventComplete> returns True when the specified event is complete; False otherwise. It is never safe to start another SPX event using the same <SPXEvent> variable until the previous event is complete. A separate <SPXRec> variable may be used to start overlapping SPX calls.

Once the function returns True, <CompletionCode> returns a status code:

- \$00 Event completed successfully.
- \$EC Connection terminated by remote partner.
- \$ED Target failed to answer.
- \$EE Invalid connection ID in <SPXEvent> variable.
- \$EF Local connection table is full.
- \$FC Event cancelled.
- \$FD Malformed packet. (too small, too large, incorrect fragment size or count)
- \$FE The socket table is full.
- \$FF Underlying socket not open. (NETWARE internal error)

While polling to determine whether an SPX event is complete, the application can perform other useful work. SPX requires only that a call to <IPXRelinquish> be made regularly so that it can check for incoming messages. A typical polling loop would be:

```
... call an SPX routine ...  
repeat  
  {application does some work}  
  IPXRelinquish;  
  {application does more work}  
until SPXEventComplete(SPXEvent, CompletionCode);
```

<SPXEventComplete> itself calls <IPXRelinquish>, so another call is needed only if the application is performing many tasks between calls.

See Also

IPXEventComplete

IPXRelinquish

SPXListenCount

Declaration

```
function SPXListenCount(var SPXEvent : SPXRec) : Byte;
```

Purpose

Returns the number of event control blocks that are currently listening for incoming messages.

Description

The number returned will be at most the number of blocks specified when the connection was established.

This function is provided primarily for debugging purposes. An application is responsible for resubmitting each control block to the free pool by calling <SPXReplenishPool> after each message is received. Therefore the number of listening blocks should remain constant except for temporary reductions. If the number decreases over time, the most probable cause is that the application is not resubmitting the blocks. If incoming messages arrive faster than the receiver can process them, the proper solution is to specify a larger event pool when the connection is established.

Example

```
if SPXListenCount(SPXEvent) = 0 then begin
  Writeln('Error: event control block pool exhausted');
  SPXTerminateConn(SPXEvent);
end;
```

Shows safety checking code that an application might use to make sure that SPX communications are correctly initialized.

See Also

SPXEstablishConn

Declaration

```
function SPXListenForConn(var SPXEvent : SPXRec;  
    Socket : Word;  
    WaitForCompletion : Boolean;  
    MaxDataSize : Word;  
    NumPoolECBs : Byte;  
    PoolESR : Pointer) : Byte;
```

Purpose

Listens for an SPX connection from a sending partner.

Description

This function is the counterpart of the <SPXEstablishConn> function. For a connection to be established, one side must listen with <SPXListenForConn>, and the other side must call with <SPXEstablishConn>.

<SPXListenForConn> uses the <SPXEvent> parameter as a work area, similar to the way that file operations use a file variable. You don't need to initialize this parameter. However, remember that <SPXEvent> must be continuously accessible as long as the SPX connection is active. It's safest to make the variable passed in the <SPXEvent> parameter a global variable.

The caller must specify a <Socket> number for the transmission. Sender and receiver must agree on a socket number (this is how SPX distinguishes between multiple incoming requests for a connection). See <IPXOpenSocket> for information regarding socket numbers. <SPXListenForConn> will open the socket if it isn't already open.

When <WaitForCompletion> is set to False, <SPXListenForConn> returns immediately after submitting the request to establish a connection. In this case, the calling application may continue with other tasks, but it must poll the <SPXEventComplete> function to determine when the event has been completed. If <WaitForCompletion> is True, <SPXListenForConn> itself polls until the event is complete. The application should then call <SPXEventComplete> to determine the outcome of the event.

<MaxDataSize> specifies the size of the largest data packet to be received on the SPX connection. If this value is too small, incoming messages will overwrite memory with unpredictable consequences. The largest possible packet is 534 bytes.

<NumPoolECBs> must always be at least two. One of the event control blocks is temporarily used to receive NetWare's acknowledgement of the connection. This block later becomes available for the management of application messages. The second block is needed for reliable management of incoming messages: if the first block is tied up while the application processes one message, the second block is available to receive another. Typically, two event control blocks are adequate. See the introduction to this section for further discussion.

SPXListenForConn

<SPXListenForConn> allocates heap space to store the event control blocks and data packets. Space required is

`<NumPoolECBS>*(106+<MaxDataSize>)`

The value 106 is `SizeOf(PoolECB)`, a lower level type defined in the NETWARE unit.

<PoolESR> is a pointer to an optional user-supplied event service routine. Passing a value of Nil deactivates this feature. Specifying a <PoolESR> would be desirable only when a high-performance, real-time, SPX connection is required. In this case the event service routine gains control immediately after each message is received, avoiding the need for a polling loop that watches for incoming data.

An event service routine must be declared exactly like the following:

```
($F+)
procedure SPXESR(var SPXEvent : SPXRec;
                  CompletionCode : Byte;
                  PoolIndex : Byte;
                  DataType : Byte;
                  DataPtr : Pointer);

begin
  ...
end;
($F-)
```

The routine must be compiled with the far model, and may not be nested inside of any other procedures or functions. The application passes the address of the routine (@SPXESR) in the <PoolESR> parameter. The routine gains control under effectively the same conditions as a hardware interrupt service routine: CPU interrupts are off at entry; non-reentrant DOS functions may not be called safely; the time spent in the routine must be strictly limited; and stack space at entry is not known but can be presumed to be small, perhaps less than 100 bytes. Turbo Pascal's stack checking should be deactivated with the {\$S-} directive to avoid problems.

Before calling the user-defined <PoolESR>, NETWARE's lower level interrupt handler saves the CPU registers and extracts important parameters from the event control block. These parameters are passed to the user routine.

SPXEvent	High level data structure associated with this connection.
CompletionCode	Status of the listen event that just occurred.
PoolIndex	Index of the control block for this event.
DataType	Type of packet received if event was successful.
DataPtr	Pointer to the data packet.

<CompletionCode> contains zero to indicate successful receipt of a data packet. See <SPXListenPooled> for a description of other error codes. <PoolIndex> must be used as a parameter to <SPXReplenishPool> in order to make the event control block available for another message. <DataType> is a user-defined value

that distinguishes the type of packet. It is just the value specified by the sender when calling <SPXSend>. And <DataPtr> is a pointer to the packet itself.

The simplest useful event handler would contain the following.

```
{ $F+ }
procedure SPXESR(var SPXEvent : SPXRec;
                  CompletionCode : Byte;
                  PoolIndex : Byte;
                  DataType : Byte;
                  DataPtr : Pointer);
begin
  if CompletionCode = 0 then begin
    {Process data of type DataType at DataPtr^}
    ...
  end else
    {Set an error flag} ;
    {Return the event control block to the free pool}
    SPXReplenishPool(SPXEvent, PoolIndex);
end;
{ $F- }
```

The event service routine may enable interrupts; it need not restore them. It may also switch to a different (larger) stack, but it must switch back to the original stack before returning. (Turbo Professional's SwapStackAndCall routine is ideal for this.) The event service routine may call any other IPX or SPX service with the exception of <IPXCloseSocket>.

SPX event service routines are a very advanced feature of NetWare. You probably won't need them, but we've made them accessible if you do.

The SPX "connection ID" eventually established by <SPXListenForConn> is stored in the <SPXEvent> variable.

A status code is returned in function result:

```
$00 Event submitted successfully.
$EF Local connection table is full.
$FE The socket table is full.
```

Example

See the introduction to this section as well as MESEXAMP.PAS.

See Also

SPXEstablishConn

SPXListenPooled

SPXListenPooled

Declaration

```
function SPXListenPooled(var SPXEvent : SPXRec;  
                        var CompletionCode : Byte;  
                        var PoolIndex : Byte;  
                        var DataType : Byte;  
                        var DataPtr : Pointer) : Boolean;
```

Purpose

Receives an SPX message from an established connection.

Description

After establishing an SPX connection, the listening station calls `<SPXListenPooled>` repeatedly until it returns True. Then the application can evaluate the returned parameters to determine the status and contents of the message.

`<CompletionCode>` always returns 0 to indicate that the message was received successfully. In the event of an error, it returns a code as shown below. The `<DataType>` and `<DataPtr>` parameters are valid only when `<CompletionCode>` is zero.

`<PoolIndex>` is the index of the event control block for this event; it is valid event `<CompletionCode>` is not zero.

`<DataType>` returns the type of data packet received. It is just the value specified when the sender called `<SPXSend>`.

`<DataPtr>` is a pointer to the data packet. The NETWARE routines have no knowledge of the contents of the packet; therefore the returned value is an untyped pointer. The application must typecast the pointer to refer to a specific user-defined type. The application also must copy the data from `<DataPtr>` to another location or else it must completely process the data before calling `<SPXReplenishPool>` to resubmit the event control block. The packet storage area at `<DataPtr>` may be overwritten by the next message.

Once the function result returns True, `<CompletionCode>` the following values:

- \$00 Message received successfully.
- \$EC Connection terminated by remote partner.
- \$EE Invalid connection ID in `<SPXEvent>` variable.
- \$FC Event cancelled.
- \$FD Malformed packet. (too small, too large, incorrect fragment size or count)
- \$FF Underlying socket not open. (NETWARE internal error)

Example

See the introduction to this section and MESEXAMP.PAS.

See Also

SPXListenForConn
SPXSend

SPXReplenishPool

SPXReplenishAll / SPXReplenishPool

Declaration

```
procedure SPXReplenishAll(var SPXEvent : SPXRec);
```

Purpose

Reactivates all event control blocks for listening.

Description

The NETWARE SPX routines automatically activate all control blocks when the connection is first established. <SPXReplenishAll> needs to be called only after one or more packets have been received.

Generally, it is safer to call <SPXReplenishPool> than to call <SPXReplenishAll>. If a message is received immediately before calling <SPXReplenishAll>, it will be lost.

See Also

SPXReplenishPool

Declaration

```
procedure SPXReplenishPool(var SPXEvent : SPXRec; PoolIndex : Byte);
```

Purpose

Reactivates a specific event control block for listening.

Description

The NETWARE SPX routines automatically activate all control blocks when a connection is first established. <SPXReplenishPool> needs to be called only after a packet has been received. <PoolIndex> is the value returned by <SPXListenPooled> when the message was received.

Example

See the introduction to this section and MESEXAMP.PAS.

See Also

SPXListenForConn

SPXListenPooled

SPXSend

Declaration

```
procedure SPXSend(var SPXEvent : SPXRec;  
                  WaitForCompletion : Boolean;  
                  DataType : Byte;  
                  DataPacketSize : Word;  
                  var DataPacket);
```

Purpose

Sends a data packet using SPX services.

Description

<SPXEvent> must have been initialized previously by a call to <SPXEstablishConn>.

The <SPXSend> event is asynchronous in the same sense that establishing the connection is. Therefore, if <WaitForCompletion> is False, an <SPXSend> must be followed by repeated calls to <SPXEventComplete>. Similarly, the <DataPacket> variable must remain static until the <SPXSend> is complete.

The packet is any user-defined data type up to 534 bytes in size. If packets of varying types are being sent, the application should use the <DataType> field to indicate the type to the receiver. Valid data types are in the range \$00..\$FD. SPX reserves the last two data types for itself.

If, for a given packet type, the length of the packet will vary, the application should pass the actual length of the packet in the first two bytes of the packet itself so that the receiver can determine the length. The <DataPacketSize> parameter is not available to the receiver. <DataPacketSize> may be in the range 0..534.

Example

See the introduction to this section and MESEXAMP.PAS.

See Also

SPXEstablishConn

SPXListenPooled

SPXServicesAvail / SPXTerminateConn

Declaration

```
function SPXServicesAvail(var Version : Word; var MaxSPXConn : Word;  
    var AvailSPXConn : Word) : Boolean;
```

Purpose

Determines if SPX services are available and returns version information if so.

Description

Any application that will use SPX services must call this routine first to assure that they are available. The function returns True if SPX is available. <Version> is the SPX version number with the major part in the high byte. <MaxSPXConn> is the maximum number of SPX connections possible. <AvailSPXConn> is the currently available number of SPX connections.

Example

```
if not SPXServicesAvail(Version, MaxConn, AvailConn) then begin  
    Writeln('SPX not available'); Halt;  
end;  
Writeln('SPX version ', Version shr 8, '.', Version and $FF);  
Writeln('Connections ', MaxConn, ' maximum, ', AvailConn, ' available');
```

Checks for SPX and reports on SPX parameters if available.

See Also

IPXServicesAvail

Declaration

```
procedure SPXTerminateConn(var SPXEvent : SPXRec);
```

Purpose

Terminate an existing SPX connection.

Description

<SPXTerminateConn> sends a packet to the other end of the connection indicating that the connection should be broken. SPX automates the receiver's acknowledgement of the message and also completes the disconnection process. The associated connection ID is returned to the free pool. The underlying IPX socket is also closed. Heap space allocated by <SPXEstablishConn> or <SPXListenForConn> is deallocated. Note that both the sender and the receiver must call <SPXTerminateConn> in order to deallocate both station's heap space. No harm is done (to the network) by closing the connection twice.

If the <SPXEvent> variable was non-static (local or allocated on the heap), it is safe to deallocate it only after calling <SPXTerminateConn>.

See Also

SPXAbortConn

SPXEstablishConn

B. NETBIOS: NetBIOS Message Passing

NetBIOS stands for Network Basic Input/Output System. It is an application programming interface (API) that supports data exchange between workstations on a network. NetBIOS was introduced by IBM in 1984, and is in widespread use today. It has become an industry standard protocol and has been implemented by most modern microcomputer network operating systems. As such, a program written using NetBIOS services will run unmodified on a wide variety of networks.

The NETBIOS unit implements the NetBIOS services for use in Turbo Pascal 4.0, 5.0, 5.5, and 6.0 programs. This unit isolates the programmer from the details of issuing the NetBIOS function calls. It is designed to make writing NetBIOS-compatible Pascal programs easy. (Note that when we spell NETBIOS in uppercase characters we are referring to the unit supplied with B-Tree Filer; the mixed case spelling NetBIOS refers to the operating system extension itself.)

Unlike the routines in the NetWare unit, the NetBIOS unit's routines are largely independent of the network operating system. In other words, they will work on any network that implements NetBIOS compatible functions. (Novell supplies a NetBIOS emulator with all versions of NetWare. Note, however, that each workstation must load additional programs--typically NETBIOS.EXE and INT2F.COM--to activate the emulator.)

In contrast to NetWare, the NetBIOS interface is compact and focused only on station to station communications. The complete NetBIOS API defines only 19 functions, and the NETBIOS unit implements and documents almost all of them.

A fundamental part of NetBIOS programming is the concept of a "name". Each workstation can have up to 17 names, each 16 bytes long. One of these names is the "permanent node name". The permanent node name is the physical network adapter card's own unique signature. It is programmed into the hardware and cannot be changed by the application. Because this name always exists, it is sometimes convenient to refer to it during other NetBIOS calls.

Each station's adapter card (or NetBIOS emulator) also supports a "local name table" that holds up to 16 software-selectable names. Each may be a "unique name", which the station reserves for its exclusive use on the network, or a "group name", which other stations can share. Case is significant when comparing NetBIOS names. When a name is added to the table, the NetBIOS broadcasts its intentions to claim the name to all other stations on the network. If a unique name was requested, and another station already has stored the same name in its local table, the request fails. When a name is successfully added to the name table, NetBIOS returns the position

in the table where it resides. This "name number" is used by many NetBIOS commands as a quick way of referring to a name that's known to be in the table.

The simplest way you can use NetBIOS to communicate from station to station is with a "datagram". A datagram is just a small block of raw data. The maximum size of a datagram is implementation specific, but the most common limit is 512 bytes. A datagram message can be sent to a unique name, a group name, or everyone on the network (with a broadcast datagram). Datagrams impose very little overhead, and are quite easy to use. However, like NetWare's IPX services, datagrams aren't automatically acknowledged by the receiver's NetBIOS and they are lost if the destination isn't ready to receive them.

A second, more robust, NetBIOS protocol is called a "session". Session packets are automatically acknowledged upon receipt, providing for guaranteed delivery of the data. Sessions can also transfer larger chunks of data--up to 64K, or even larger by using advanced techniques. Of course, with this reliability comes extra overhead. Another limitation of sessions relative to datagrams is that they are essentially one-to-one connections--broadcasts aren't allowed.

The NETBIOS unit supports both datagrams and sessions. The following list summarizes the unit's functions:

- determine whether NetBIOS is installed
- reset the network adapter
- add a unique name to the local name table
- add a group name to the local name table
- remove a name from the local name table
- send a datagram
- receive a datagram
- establish a session
- send a session message
- receive a session message
- close a session

NetBIOS calls are issued in one of two ways. In both cases, the register pair ES:BX points to an initialized record called a NetBIOS Control Block (NCB). Then either an int \$5C, or an int \$21 (DOS) function \$2A is issued. The NETBIOS unit initializes the NCB and performs the interrupt for you. (NETBIOS uses the int \$5C rather than the int \$21 method.)

Many NetBIOS calls can operate in a "no-wait" mode. This means that the call returns immediately after the request is submitted and doesn't wait for completion (similar to the way IPX performs under NetWare). In this case, the NCB variable passed as an argument to the NETBIOS routine must remain undisturbed in memory until the call has been completed. Care must be taken not to reuse the NCB and, if the NCB is a local variable, not to exit the routine where it was declared until the NetBIOS function is done. When no-wait calls are used, the safest methodology is to declare one NCB per NetBIOS request, declare NCBs as global variables, and check the <CmdComplete> field of the NCB before reusing it.

The following references on NetBIOS are useful:

C Programmer's Guide to NetBIOS

W. David Schwaderer, Howard W. Sams & Co., 1988.

Inside NetBIOS

J. Scott Haugdahl, Architecture Technology Corp, Minneapolis, MN.

NetBIOS Applications Development Guide

IBM Manual #68X2270.

Examples

We'll first consider the low level datagram services. The sender and receiver must agree on the names by which they refer to each other. In some cases, it is appropriate to extract these names from the workstation environment; for our example, we'll just use arbitrary names coded into the application. See Section 2.F for a discussion of choosing names in a network environment.

The sender's side of a datagram message looks as follows:

```
const
  SenderName : NbNameStr = 'SenderName';
  ReceiverName : NbNameStr = 'ReceiverName';

function SendOneDatagram(MsgLen : Word;
                        var Msg) : Byte;
var
  Status : Byte;
  NameNumber : Byte;
  N : NCB;
begin
  (Add name to refer to)
  Status := NetBiosAddName(SenderName, NameNumber);
  if Status <> 0 then begin
    (Problem with sender name)
    SendOneDatagram := Status;
    Exit;
  end;
  (Send the message)
  SendDatagram(N, NameNumber, ReceiverName, False, MsgLen, Msg);
```

```

{Wait for confirmation that it was sent}
while N.CmdComplete = NbeCommandPending do
  if KeyPressed then begin
    {User got impatient}
    Status := NetBiosDeleteName(SenderName);
    SendOneDatagram := 2;
    Exit;
  end;
  Status := NetBiosDeleteName(SenderName);
  {Return status, zero for success}
  SendOneDatagram := N.RetCode;
end;

```

This routine sets up and sends a single datagram, which may be as large as 512 bytes. Because SendOneDatagram adds and deletes the sender name for each message it would waste time for multiple datagrams, but it does neatly encapsulate the sending process. A real application would probably add the reference name once when it started and delete it prior to exiting.

The receiver's side looks like this:

```

function ReceiveOneDatagram(MaxLen : Word;
                           var Msg) : Byte;
var
  Status : Byte;
  NameNumber : Byte;
  N : NCB;
begin
  {Add name to refer to}
  Status := NetBiosAddName(ReceiverName, NameNumber);
  if Status <> 0 then begin
    {Problem with receiver name}
    ReceiveOneDatagram := Status;
    Exit;
  end;
  {Listen for the message}
  ReceiveDatagram(N, NameNumber, False, MaxLen, Msg);
  {Wait to receive message}
  while N.CmdComplete = NbeCommandPending do
    if KeyPressed then begin
      {User got impatient}
      Status := NetBiosDeleteName(ReceiverName);
      ReceiveOneDatagram := 2;
      Exit;
    end;
    Status := NetBiosDeleteName(ReceiverName);
    {Return status, zero for success}
    ReceiveOneDatagram := N.RetCode;
  end;
end;

```

Note that the receiver does not find out exactly how long the message was. If the length exceeds MaxLen, NetBIOS returns an error code and the receiver can call ReceiveOneDatagram again to obtain the overflow.

NetBIOS session services are more reliable and allow longer messages than the datagram services. Before messages can be sent and received, a session must be established. The sender establishes its side of the session as follows:

```
function CallSession(var Session : Byte) : Byte;
var
  Status : Byte;
  NameNumber : Byte;
begin
  {Add name to refer to}
  Status := NetBiosAddName(SenderName, NameNumber);
  if Status <> 0 then begin
    {Problem with sender name}
    CallSession := Status;
    Exit;
  end;
  {Open the sender's side of session}
  repeat
    Status := NetBiosCall(ReceiverName, SenderName, 5, 5, Session);
    if Status <> 0 then
      if KeyPressed then begin
        {User got impatient}
        Status := NetBiosDeleteName(SenderName);
        CallSession := 2;
        Exit;
      end;
    until Status = 0;
    {Always return success if we get here}
    CallSession := Status;
  end;
```

The two values of 5 passed to <NetBiosCall> specify that timeouts on subsequent sends and receives will take 2.5 seconds each (5 one-half second intervals).

The receiver's side of the session is similar:

```
function ListenSession(var Session : Byte) : Byte;
var
  Status : Byte;
  NameNumber : Byte;
begin
  {Add name to refer to}
  Status := NetBiosAddName(ReceiverName, NameNumber);
  if Status <> 0 then begin
    {Problem with receiver name}
    ListenSession := Status;
    Exit;
  end;
  {Open the receiver's side of session}
  repeat
    Status := NetBiosListen(SenderName, ReceiverName, 5, 5, Session);
    if Status <> 0 then
      if KeyPressed then begin
        {User got impatient}
```

```

        Status := NetBiosDeleteName(ReceiverName);
        ListenSession := 2;
        Exit;
    end;
until Status = 0;
{Return success if we get here}
ListenSession := Status;
end;

```

Once both sides of the session are established, either station can send or receive. Each refers to the session number returned when the session was established. A message is sent as follows:

```

Status := NetBiosSend(Session, MsgLen, Msg);

```

Sessions can send and receive messages up to 64K bytes long. Any non-zero Status value indicates an error. Similarly, a message is received with:

```

Status := NetBiosReceive(Session, MaxLen, Msg);

```

Neither of these calls allows the user to abort before the predetermined timeout period.

Finally, to terminate the session, either side can call the following routine, passing the appropriate name to delete from the name table.

```

procedure TerminateSession(NameToDelete : NbNameStr);
var
    Status : Byte;
begin
    Status := NetBiosHangUp(Session);
    Status := NetBiosDeleteName(NameToDelete);
end;

```

Complete examples of these concepts can be found in MESEXAMP, NSEND, and NRECEIVE.

In the session examples just shown, the NETBIOS unit makes calls that wait for completion of NetBIOS events such as opening a session. Use of the no-wait option is important under certain NetBIOS implementations (PC-LAN in particular) because the network will never time out (when the receiver never answers, for example) even if finite timeout parameters have been specified. The NSEND and NRECEIVE demonstration programs use the no-wait routines, which are more robust for use on a variety of NetBIOS networks.

The following small sample program shows how to send a session message using the no-wait services provided by NETBIOS:

```

program ExNBSend;
uses
  Crt, NetBIOS;
const
  SenderName  : NbNameStr = 'SenderName';
  ReceiverName : NbNameStr = 'ReceiverName';
  Msg : String = 'Sample Data';
var
  N      : NCB;
  SessionLSN : Byte;
  RetCode  : Byte;
  NameNum   : Byte;

  procedure WaitForCompletion(var N : NCB; var RetCode : Byte);
  begin
    while not NetBiosCmdCompleted(N, RetCode) do
      if KeyPressed then
        {User got impatient}
        if CancelRequest(N) = 0 then ;
      end;
    end;
  begin
    {Add name to refer to}
    if NetBiosAddName(SenderName, NameNum) <> 0 then begin
      Writeln('Error adding sender name');
      Halt;
    end;
    {Open the sender's side of the session}
    NetBiosCallNoWait(ReceiverName, SenderName, 5, 5, Nil, N);
    WaitForCompletion(N, RetCode);
    if RetCode = 0 then begin
      {Session established}
      SessionLSN := N.LSN;
      {Send the message}
      NetBiosSendNoWait(SessionLSN, SizeOf(Msg), Nil, N, Msg);
      WaitForCompletion(N, RetCode);
    end else
      SessionLSN := 255;
    if RetCode <> 0 then
      Writeln('Error ', RetCode, ' transmitting message')
    else
      Writeln('Message transmitted');
    if NetBiosHangUp(SessionLSN) <> 0 then
      Writeln('Error terminating session');
    if NetBiosDeleteName(SenderName) <> 0 then
      Writeln('Error removing sender name');
  end.

```

This example is quite similar to CallSession shown previously. The main difference is that the code in WaitForCompletion loops much more frequently than the corresponding loop in CallSession. The function <NetBiosCmdCompleted> returns False until the NetBIOS indicates (by modifying the <NCB>) that the event has been completed. In the meantime, WaitForCompletion continuously checks the keyboard so that the user can abort the operation if desired. After the event is

cancelled by calling <CancelRequest>, <NetBiosCmdCompleted> returns True, and RetCode equals <NbeCommandCancelled> (\$9B).

Here's the corresponding code for the receiver's end. (Procedure WaitForCompletion is the same as on the sender's side.)

```
program ExNBReceive;
uses
  Crt, NetBIOS;
const
  SenderName : NbNameStr = 'SenderName';
  ReceiverName : NbNameStr = 'ReceiverName';
var
  N : NCB;
  SessionLSN : Byte;
  RetCode : Byte;
  NameNum : Byte;
  Msg : String;

  procedure WaitForCompletion(var N : NCB; var RetCode : Byte);
  ...

begin
  {Add name to refer to}
  if NetBiosAddName(ReceiverName, NameNum) <> 0 then begin
    Writeln('Error adding receiver name');
    Halt;
  end;
  {Open the receiver's side of the session}
  NetBiosCallNoWait(SenderName, ReceiverName, 5, 5, Nil, N);
  WaitForCompletion(N, RetCode);
  if RetCode = 0 then begin
    {Session established}
    SessionLSN := N.LSN;
    {Send the message}
    NetBiosReceiveNoWait(SessionLSN, SizeOf(Msg), Nil, N, Msg);
    WaitForCompletion(N, RetCode);
  end else
    SessionLSN := 255;
  if RetCode <> 0 then
    Writeln('Error ', RetCode, ' receiving message')
  else
    Writeln('Message received ', Msg);
  if NetBiosHangUp(SessionLSN) <> 0 then
    Writeln('Error terminating session');
  if NetBiosDeleteName(ReceiverName) <> 0 then
    Writeln('Error removing receiver name');
end.
```

This code is very similar to the sender's side.

Instead of using a routine like WaitForCompletion, it's possible to use a "post-event" routine. In this case, the NetBIOS will automatically generate a call to a

specified routine when the event is complete. In the meantime, the application can go on about its business (as long as the <NCB> variable remains in scope throughout the time while the event is in progress). All of the "no-wait" NetBIOS routines take a parameter that specifies the address of a post-event routine. With Nil in this position, no such routine is used. If an address is passed in this position, it must point to a Turbo Pascal interrupt procedure, which is declared as follows:

```
procedure SamplePostRoutine(Flags, CS, IP,
                             AX, BX, CX, DX, SI, DI, DS, ES, BP : Word);
interrupt;
```

Since this routine is completely analogous to a hardware interrupt handler, it must be written with the same caution. DOS services may not be available when the routine is activated (since DOS is not reentrant); the stack size may be very limited; time-critical foreground processes may be in progress at the time of the call.

When the post-event routine is activated, ES:BX points to the NetBIOS control block (NCB) that was completed, and AL contains the NetBIOS completion code.

Constants

```
DefaultAdapterNum : Byte = 0;
```

The network adapter number used for all calls in the NETBIOS unit. Allowed values are 0 or 1, with 0 being the primary adapter.

```
NbCall           = $10;
NbListen         = $11;
NbHangup        = $12;
NbSend          = $14;
NbReceive       = $15;
NbReceiveAny    = $16;
NbChainSend     = $17;
NbSendDatagram  = $20;
NbReceiveDatagram = $21;
NbSendBDatagram = $22;
NbReceiveBDatagram = $23;
NbAddName       = $30;
NbDeleteName    = $31;
NbResetWaitOnly = $32;
NbCancelWaitOnly = $35;
NbAddGroupName = $36;
NbSendNoAck     = $71;
NbChainSendNoAck = $72;
NbFindName      = $78;
NbInvalidCommand = $7F; {Invalid command for testing NetBIOS installation}
```

Values used in the Command field of the NCB.

```
NbcCommandPending = $FF;
```

When stored in the <CmdComplete> field of an <NCB>, indicates that a command is still pending.

NbeSuccess	= \$00;
NbeInvalidBufferLength	= \$01;
NbeInvalidCommand	= \$03;
NbeTimedOut	= \$05;
NbeIncomplete	= \$06;
NbeLocalNoAckFailed	= \$07;
NbeInvalidLsn	= \$08;
NbeNoResourceAvail	= \$09;
NbeSessionClosed	= \$0A;
NbeCommandCancelled	= \$0B;
NbeDuplicateName	= \$0D;
NbeNameTableFull	= \$0E;
NbeNameHasActive	= \$0F;
NbeLocalSessionTableFull	= \$11;
NbeSessionNoListen	= \$12;
NbeIllegalNameNumber	= \$13;
NbeCannotFindName	= \$14;
NbeNoAnswer	= \$14;
NbeInvalidName	= \$15;
NbeNameInUseOnRemote	= \$16;
NbeNameDeleted	= \$17;
NbeSessionAbnormal	= \$18;
NbeNameConflict	= \$19;
NbeIncompatibleDevice	= \$1A;
NbeInterfaceBusy	= \$21;
NbeTooManyCommands	= \$22;
NbeInvalidLanA	= \$23;
NbeCompletedWhileCancel	= \$24;
NbeReservedName	= \$25;
NbeNotValidCancel	= \$26;
NbeSystemError	= \$40;
NbeHotCarrierFromRemote	= \$41;
NbeHotCarrier	= \$42;
NbeNoCarrier	= \$43;
NbeUnexpectedAdaptClose	= \$FD;

NetBIOS error codes returned in the <RetCode> field of an <NCB>. See the routines in this chapter for more information about how they apply to each NetBIOS command. Many NetBIOS commands can generate errors \$21 and \$22. When this occurs, the appropriate response is to wait a while and try again. Commands can also generate errors \$40-\$FE. These errors indicate a hardware failure or an internal NetBIOS error. The appropriate response is to halt the application and repair the hardware.

NbNameMax = 16;

The maximum length of a NetBIOS name.

NoWait = \$80;

Bit mask used to set the high bit of values stored in the <Command> field of an <NCB> to indicate that NetBIOS is not to wait for the completion of an event. In the NETBIOS unit, this constant is used optionally within the commands to send and receive datagrams.

Types

```
CallNameType = array[1..NbNameMax] of Char;
```

Internal format used by NetBIOS to store names.

```
NCB =  
  record  
    Command : Byte;  
    RetCode : Byte;  
    LSN : Byte;  
    NameNum : Byte;  
    Buffer : Pointer;  
    BufLen : Word;  
    CallName : CallNameType;  
    Name : CallNameType;  
    RTO : Byte;  
    STO : Byte;  
    PostRoutine : Pointer;  
    LanaNum : Byte;  
    CmdComplete : Byte;  
    Reserved : array[1..14] of Byte;  
  end;
```

The NetBIOS control block. This 64 byte record structure is the heart of all NetBIOS requests. Here is a brief description of all the fields:

Command	specifies which NetBIOS command to execute
RetCode	final result of the command. Valid only if CmdComplete <> \$FF
LSN	local session number associated with command
NameNum	name number associated with command
Buffer	pointer to data buffer for command
BufLen	length of the data buffer
CallName	usually, remote name associated with command
Name	local name associated with command
RTO	receive timeout limit, in 0.5 second units
STO	send timeout limit, in 0.5 second units
PostRoutine	pointer to routine executed upon completion of no-wait command
LanaNum	network adapter to use, typically 0
CmdComplete	\$FF if command pending, any other value when complete
Reserved	used internally by NetBIOS

<PostRoutine> is a pointer to a routine supplied by the programmer. See the discussion of no-wait services earlier in this section for more information.

```
NbNameStr = String[NbNameMax];
```

String type used for NetBIOS names.

Declaration

```
function CancelRequest(var N : NCB) : Byte;
```

Purpose

Cancels a pending NetBIOS request.

Description

<N> is the <NCB> of the command that is to be cancelled. This routine may be used to cancel the datagram services <ReceiveDatagram> and <ReceiveBDatagram> when the <Wait> option is False. It may also be used to cancel events submitted by <NetBiosSendNoWait>, <NetBiosReceiveNoWait>, <NetBiosCallNoWait>, and <NetBiosListenNoWait>.

You can never cancel the following NetBIOS commands: <CancelRequest>, <NetBiosAddName>, <NetBiosAddGroupName>, <NetBiosDeleteName>, <NetBiosHangUp>, <ResetAdapter>, <SendDatagram>, and <SendBDatagram>.

A status code is returned in function result:

- \$00 Command cancelled.
- \$24 Command completed while cancel occurring.
- \$26 Command not valid to cancel.

See Also

SendDatagram

ReceiveDatagram

Declaration

```
procedure ClearNCB(var N : NCB);
```

Purpose

Initializes an <NCB> to all zeros.

Description

This is always required before using an NCB. The higher level routines in the NETBIOS unit call this procedure to initialize all <NCB>'s they use. It is interfaced in case the programmer needs to make a direct call to the NetBIOS.

See Also

NetBiosRequest

NetBiosAddGroupName

Declaration

```
function NetBiosAddGroupName(NameToAdd : NbNameStr;  
                             var NameNumber : Byte) : Byte;
```

Purpose

Adds a group name to the local NetBIOS name table.

Description

Other nodes may be using the same name as a group name, but not as a unique name. NetBIOS repeatedly broadcasts this name across the network. If no conflicting response is received after an adequate number of retries has been attempted, <NetBiosAddGroupName> succeeds. This function does not return until the command is complete.

<NameToAdd> is any ASCII string up to 16 characters long. IBM imposes several restrictions on the name, however. It may not begin with a null (#0), asterisk (*), or the three letters 'IBM'. The sixteenth character may not be ASCII #31.

<NetBiosAddGroupName> pads the string with nulls to reach 16 characters. Case is significant when comparing NetBIOS names.

<NameNumber> returns the local name table slot assigned to the name. This number is used in calling the datagram routines.

A status code is returned in function result:

- \$00 Name added successfully.
- \$0D Duplicate name in local table.
- \$0E Name table is full.
- \$15 Invalid name.
- \$16 Name in use on another station.
- \$19 Name conflict detected. (duplicate names elsewhere)

See Also

NetBiosAddName

NetBiosDeleteName

Declaration

```
function NetBiosAddName(NameToAdd : NbNameStr;  
                        var NameNumber : Byte) : Byte;
```

Purpose

Adds a unique name to the local NetBIOS name table.

Description

NetBIOS repeatedly broadcasts this name across the network. If no matching response is received after an adequate number of retries has been attempted, <NetBiosAddName> succeeds. This function does not return until the command is complete.

<NameToAdd> is any ASCII string up to 16 characters long. IBM imposes several restrictions on the name, however. It may not begin with a null (#0), asterisk (*), or the three letters 'IBM'. The sixteenth character may not be ASCII #31.

<NetBiosAddName> pads the string with nulls to reach 16 characters. Case is significant when comparing NetBIOS names.

<NameNumber> returns the local name table slot assigned to the name. This number is used in calling the datagram routines.

A status code is returned in function result:

- \$00 Name added successfully.
- \$0D Duplicate name in local table.
- \$0E Name table is full.
- \$15 Invalid name.
- \$16 Name in use on another station.
- \$19 Name conflict detected. (duplicate names elsewhere on network)

See Also

NetBiosAddGroupName
NetBiosCall

NetBiosDeleteName
NetBiosCallNoWait

NetBiosCall

Declaration

```
function NetBiosCall(RemoteName, LocalName : NbNameStr;  
    SendTimeOut, RecTimeOut : Byte;  
    var SessionLSN : Byte) : Byte;
```

Purpose

Call another station to initiate a session.

Description

The remote station must call <NetBiosListen> before the local station calls <NetBiosCall>. To make a successful connection, each command is generally called within a loop (which should also provide an opportunity for the user to abort). Once the session has been created, either station can send or receive messages.

<LocalName> specifies an existing name in the local adapter's name table. <RemoteName> specifies a name in the table of the listener. Both ends must agree on the names in order for a session to be created. Generally, the names should be unique, since a session cannot connect more than two stations. It is also possible for both ends of the session to be on a single workstation. <SendTimeOut> and <ReceiveTimeOut> specify how long the NetBIOS will retry, measured in half-second increments, when later sending and receiving messages. A value of zero means to retry forever. Note that some NetBIOS implementations ignore these parameters. Use <NetBiosCallNoWait> in this case.

Upon successful completion of this routine, a local session number is returned in <SessionLSN>. This number is used in later calls to send or receive messages, and to disconnect the session. A status code is returned in the function result:

- \$00 Session established.
- \$05 Command timed out.
- \$09 Remote session table full.
- \$11 Local session table full.
- \$12 No listen is outstanding.
- \$14 Cannot find name called, or no answer.
- \$15 Invalid name.
- \$18 Session ended abnormally.
- \$19 Name conflict detected. (duplicate names elsewhere on network)

The strategy for choosing NetBIOS names is similar to choosing unique socket numbers under NetWare. The easiest, and usually the best, strategy is to code the name directly into the application, but to make that name installable by the network administrator. In the unlikely event of a conflict with another application running on the network, the administrator can choose a different name.

Alternative strategies involve the use of broadcast datagrams or shared files to establish an acceptable set of NetBIOS names.

See Also

NetBiosListen

NetBiosCallNoWait / NetBiosCmdCompleted

Declaration

```
procedure NetBiosCallNoWait(RemoteName, LocalName : NbNameStr;  
    SendTimeout, RecTimeout : Byte;  
    PostEvent : Pointer;  
    var N : NCB);
```

Purpose

Call another station to initiate a session; don't wait for connection.

Description

This routine works like <NetBiosCall>, except that it returns immediately after posting the event. The application should either specify a <PostEvent> routine or should loop calling <NetBiosCmdCompleted> to see whether the session has been established. The parameter <N> is initialized by this call; its associated variable must remain valid until the event completes or is cancelled.

Example

See the introduction to this section.

See Also

CancelRequest
NetBiosCmdCompleted

NetBiosCall
NetBiosListenNoWait

Declaration

```
function NetBiosCmdCompleted(var N : NCB: var FinalRetCode : Byte) : Boolean;
```

Purpose

Determine whether no-wait event has been completed.

Description

Call this routine repeatedly after initializing any NetBIOS event in the no-wait condition. It will return True and the final status code <FinalRetCode> when the event has been completed. In the meantime the application may continue with other tasks or cancel the event if desired.

The parameter <N> is the same <NCB> that was initialized when the event was started. This variable must remain in scope and must not be reused until the event completes or is cancelled.

Calling <NetBiosCmdCompleted> is equivalent to comparing the <CmdComplete> field of the <NCB> to <NbCommandPending>. See <ReceiveDatagram> for a comparison of the two techniques.

See Also

NetBiosCallNoWait
NetBiosReceiveNoWait

NetBiosListenNoWait
NetBiosSendNoWait

NetBiosDeleteName

Declaration

```
function NetBiosDeleteName(NameToDelete : NbNameStr) : Byte;
```

Purpose

Deletes a unique or group name from the local NetBIOS name table.

Description

If a name is deleted while a session is actively using it, the actual deletion is delayed until the session is terminated, typically by a <NetBiosHangup> command. <NetBiosDeleteName> returns error code \$0F in this case. Even though the name table space is not immediately reclaimed, the name is "deregistered", so that datagrams and new sessions cannot use it.

All names added to the name table should be deleted before an application exits. NetBIOS does not do this automatically. See the demonstration programs NSEND and NRECEIVE for examples of using a Turbo Pascal exit procedure to delete the names.

A status code is returned in the function result:

- \$00 Name deleted successfully.
- \$0F Name has active sessions. Deletion is delayed.
- \$15 Invalid name.

See Also

NetBiosAddGroupName

NetBiosAddName

NetBiosHangUp / NetBiosInstalled

Declaration

```
function NetBiosHangUp(SessionLSN : Byte) : Byte;
```

Purpose

Terminates existing session specified by <SessionLSN>.

Description

The function does not return until the command is complete. Either side of a session may call this routine to terminate the session.

A status code is returned in the function result:

- \$05 Command timed out.
- \$08 Invalid local session number.
- \$0A Session already closed.
- \$0B Command was cancelled.
- \$18 Session ended abnormally.

NetBIOS will terminate any pending NetBIOS commands that use the specified session when this routine is called, although this may generate a delay before <NetBiosHangUp> returns.

See Also

NetBiosCall

NetBiosDeleteName

Declaration

```
function NetBiosInstalled : Boolean;
```

Purpose

Returns True if NetBIOS or compatible emulator is installed.

Description

A check is made to see whether the int \$5C vector contains the Nil address or points into the PC ROM BIOS. If either is the case, it can be concluded that NetBIOS is not loaded. Otherwise, an invalid NetBIOS request (\$7F) is issued. This command will return the constant <NbeInvalidCommand> in <NCB.RetCode> if NetBIOS is installed. There is a slight chance that this technique will conflict with another int \$5C handler, but since this interrupt is reserved for NetBIOS by IBM, it is quite unlikely.

Example

```
if not NetBiosInstalled then begin
  Writeln('This program requires NetBIOS communications');
  Halt;
end;
```

Halts an application if NetBIOS is not installed.

NetBiosListen

Declaration

```
function NetBiosListen(RemoteName, LocalName : NbNameStr;  
    SendTimeOut, RecTimeOut : Byte;  
    var SessionLSN : Byte) : Byte;
```

Purpose

Listen for a call that will initiate a session.

Description

The local station must call <NetBiosListen> before the remote station calls <NetBiosCall>. To make a successful connection, each command is generally called within a loop (which should also provide an opportunity for the user to abort). Once the session has been created, either station can send or receive messages.

<LocalName> specifies an existing name in the local adapter's name table.

<RemoteName> specifies a name in the table of the caller. Both ends must agree on the names in order for a session to be created. Generally, the names should be unique, since a session cannot connect more than two stations. It is also possible for both ends of the session to be on a single workstation.

<SendTimeOut> and <ReceiveTimeOut> specify how long the NetBIOS will retry, measured in half-second increments, when later sending and receiving messages. A value of zero means to retry forever. Note that some NetBIOS implementations ignore these parameters. Use <NetBiosListenNoWait> in this case.

Upon successful completion of this routine, a local session number is returned in <SessionLSN>. This number is used in later calls to send or receive messages, and to disconnect the session.

A status code is returned in the function result:

- \$00 Session established.
- \$05 Command timed out.
- \$09 Remote session table full.
- \$11 Local session table full.
- \$15 Invalid name.
- \$18 Session ended abnormally.
- \$19 Name conflict detected. (duplicate names elsewhere on network)

Example

See the introduction to this section.

See Also

NetBiosCall

NetBiosListenNoWait

Declaration

```
procedure NetBiosListenNoWait(RemoteName, LocalName : NbNameStr;  
    SendTimeout, RecTimeout : Byte;  
    PostEvent : Pointer;  
    var N : NCB);
```

Purpose

Listen to initiate a session; don't wait for connection.

Description

This routine works like <NetBiosListen>, except that it returns immediately instead of waiting for the connection to be established or for timeout to occur.

The application should either specify a <PostEvent> routine or should loop calling <NetBiosCmdCompleted> to see whether the session has been established. The parameter <N> is initialized by this call; its associated variable must remain valid until the event completes or is cancelled.

Example

See the introduction to this section.

See Also

CancelRequest	NetBiosCallNoWait
NetBiosCmdCompleted	NetBiosListen

NetBiosReceive

Declaration

```
function NetBiosReceive(SessionLSN : Byte;  
                        PacketSize : Word;  
                        var Packet) : Byte;
```

Purpose

Receives data during a session.

Description

<SessionLSN> must have been initialized by a prior call to <NetBiosCall> or <NetBiosListen>. <NetBiosReceive> waits for the command to complete, either by receiving the data or by timing out with an error.

<PacketSize> specifies the maximum size of the data packet (0..65535) and <Packet> is the buffer to receive the data. It is essential that the <Packet> variable be at least as large as the <PacketSize> parameter specifies. Otherwise, memory may be overwritten, probably leading to a program crash. If the incoming packet is larger than <PacketSize>, NetBIOS will generate an error but will not overwrite memory.

A status code is returned in the function result:

```
$00 Message received.  
$05 Command timed out.  
$06 Incomplete received message. (buffer too small)  
$08 Invalid local session number.  
$0A Session has been closed.  
$0B Command was cancelled.  
$18 Session ended abnormally.
```

If an error \$06 is returned, the receiver can request the remainder of the packet by immediately issuing another <NetBiosReceive> call.

Example

See the introduction to this section.

See Also

NetBiosReceiveNoWait

NetBiosSend

NetBiosReceiveNoWait / NetBiosRequest

Declaration

```
procedure NetBiosReceiveNoWait(SessionLSN : Byte;  
                               PacketSize : Word;  
                               PostEvent : Pointer;  
                               var N : NCB;  
                               var Packet) : Byte;
```

Purpose

Receives data during a session; does not wait for completion.

Description

This routine works like <NetBiosReceive>, except that it does not wait to receive the message before returning.

The application should either specify a <PostEvent> routine or should loop calling <NetBiosCmdCompleted> to see whether the message has been received. The parameter <N> is initialized by this call; its associated variable must remain valid until the event completes or is cancelled.

Example

See the introduction at the beginning of this section.

See Also

NetBiosReceive

NetBiosSendNoWait

Declaration

```
procedure NetBiosRequest(var N : NCB);
```

Purpose

Issues a direct NetBIOS call.

Description

This routine sets ES:BX to point to the <NCB> and then calls interrupt \$5C. The <NCB> must have been initialized previously by calling <ClearNCB> and then setting the appropriate fields. Applications won't typically need to call this low-level routine. It is interfaced in case the programmer needs to make a direct call to the NetBIOS.

This routine is implemented as an inline macro, which actually uses less code space than a far call to another routine would consume.

See Also

ClearNCB

NetBiosSend / NetBiosSendNoWait

Declaration

```
function NetBiosSend(SessionLSN : Byte; PacketSize : Word; var Packet) : Byte;
```

Purpose

Sends data during a session.

Description

<SessionLSN> must have been initialized by a prior call to <NetBiosCall> or <NetBiosListen>. <NetBiosSend> waits for the command to complete, either with an acknowledgement from the receiver or by timing out with an error.

<PacketSize> is the size of the data packet (0..65535). The data sent is passed in the parameter <Packet>. Since this is an untyped var parameter, any data type may be used when calling the routine. It is usually best to use Turbo Pascal's SizeOf() operator to specify the size of the variable, as in the following example:

```
var
  Buffer : array[1..1000] of Char;
...
Status := NetBiosSend(SessionNumber, SizeOf(Buffer), Buffer);
```

A status code is returned in the function result:

```
$00 Message sent.
$05 Command timed out.
$08 Invalid local session number.
$0A Session has been closed.
$0B Command was cancelled.
$18 Session ended abnormally.
```

See Also

NetBiosReceive

NetBiosSendNoWait

Declaration

```
procedure NetBiosSendNoWait(SessionLSN : Byte; PacketSize : Word;
  PostEvent : Pointer;
  var N : NCB;
  var Packet) : Byte;
```

Purpose

Send a session message; do not wait for completion.

Description

This routine works like <NetBiosSend>, except that it does not wait for the message to be sent before returning.

See Also

NetBiosReceiveNoWait

NetBiosSend

ReceiveBDatagram

Declaration

```
procedure ReceiveBDatagram(var N : NCB; ReceiverNameNum : Byte;  
                           Wait : Boolean; DGSize : Word; var Datagram);
```

Purpose

Receive a broadcast datagram.

Description

This command works just like <ReceiveDatagram>, but it receives only those messages sent by <SendBDatagram>. See <ReceiveDatagram> for more information.

Example

```
program ReceiveBroadcast;  
uses  
  NetBios;  
var  
  N : NCB;  
  S : String;  
begin  
  ReceiveBDatagram(N, 1, True, SizeOf(String), S);  
  if N.RetCode = 0 then  
    Writeln('Received string ', S)  
  else  
    Writeln('Error receiving broadcast');  
end;
```

A simple program to receive a broadcast datagram using name number 1 in the receiver's name table.

See Also

SendBDatagram

ReceiveDatagram

Declaration

```
procedure ReceiveDatagram(var N : NCB;  
    ReceiverNameNum : Byte;  
    Wait : Boolean;  
    DGSize : Word;  
    var Datagram);
```

Purpose

Receive a normal datagram.

Description

Datagrams received are destined for the NetBIOS name with number <ReceiverNameNum> in the local name table. Any other station may have transmitted the datagram. The transmission may have been directed to a unique name or to a group name shared by many stations. Although this command will not receive broadcast datagrams, it can be made to intercept any normal datagram if the value \$FF is specified for <ReceiverNameNum>.

The call to <ReceiveDatagram> must be pending at the time the sender transmits the message. If not, the message will be lost.

If <Wait> is True, the call to <ReceiveDatagram> does not return until a message is received. Use this option with care, because there is no time out value associated with datagram reception. In most cases, setting <Wait> to False is the better option. Then the routine returns immediately. The calling application can go ahead with other work while waiting for a message to appear. In the meantime, the NetBIOS control block <N> and the <Datagram> data buffer must remain undisturbed in memory because the NetBIOS will eventually write status and data to them.

A variable of any type may be passed in the untyped parameter <Datagram>. The maximum size of a datagram is implementation specific, but is typically 512 bytes. Be sure that the size of the buffer is at least as large as <DGSize> or NetBIOS will wipe out an undesired portion of memory.

To determine when the action is complete, the application can poll the <CmdComplete> field of the <NCB>:

```
while (N.CmdComplete = NbcCommandPending) and not TimedOut do begin  
    {Do other work}  
    {Set TimedOut True if too much time passes}  
end;  
if TimedOut then  
    {Cancel the request to receive}  
    Status := CancelRequest(N)  
else if N.RetCode <> 0 then  
    {Handle error}  
else  
    {Message received} ;
```

ReceiveDatagram

Alternatively the application can use the <NetBiosCmdCompleted> function:

```
while not(NetBiosCmdCompleted(N, RetCode) or TimedOut) do begin
  {Do other work}
  {Set TimedOut if too much time passes}
end;
if TimedOut then
  {Cancel the request to receive}
  Status := CancelRequest(N)
else if RetCode <> 0 then
  {Handle error}
else
  {Message received}
```

In this latter case, RetCode is a local variable of type Byte.

A status code is returned in the function result:

```
$00 Datagram received.
$06 Incomplete received message.
$08 Command was cancelled.
$13 Illegal name number.
$17 Name was deleted.
$19 Name conflict detected.
```

If error \$06 is received, the message was larger than <DGSize>. The remainder of the message is lost.

See Also

ReceiveBDatagram

SendDatagram

ResetAdapter / SendBDatagram

Declaration

```
function ResetAdapter(SessionCount : Byte; CommandCount : Byte) : Byte;
```

Purpose

Resets a NetBIOS adapter card.

Description

The global typed constant <DefaultAdapterNum>, which defaults to 0, specifies the adapter card to reset. This command was originally designed to reset the hardware network adapter for IBM's PC LAN; NetBIOS emulators may treat the command differently. One of its functions is to clear the local name and session tables; all NetBIOS versions should perform this action.

<SessionCount> is the number of simultaneous sessions that the current workstation will allow. (Default value is 6 for PC LAN.) <CommandCount> is the maximum number of pending commands. (Default value is 12.) Unless the NETBIOS unit's no-wait services are being used, the command count should not matter.

A status code is returned in the function result:

```
$00 Successful reset.  
$23 Invalid adapter number.
```

Declaration

```
procedure SendBDatagram(var N : NCB; SenderNameNum : Byte;  
                        Wait : Boolean; DGSize : Word; var Datagram);
```

Description

This command works just like <SendDatagram>, except that the message is sent to all nodes. Only nodes listening with <ReceiveBDatagram> will receive the message. See <SendDatagram> for more information.

Example

```
program SendBroadcast;  
uses NetBios;  
var N : NCB; S : String;  
begin  
  S := 'Data to send';  
  SendBDatagram(N, 1, True, Length(S)+1, S);  
  if N.RetCode = 0 then  
    Writeln('Transmitted string');  
end.
```

A simple program to receive a broadcast datagram using name number 1 in the receiver's name table.

See Also

ReceiveBDatagram

SendDatagram

Declaration

```
procedure SendDatagram(var N : NCB;  
    SenderNameNum : Byte;  
    ReceiverName : NbNameStr;  
    Wait : Boolean;  
    DGSize : Word;  
    var Datagram);
```

Purpose

Sends a datagram to a specified NetBIOS name.

Description

If <ReceiverName> is a unique name, the message will go to at most one station. If <ReceiverName> is a group name, the message may go to many stations at once. <ReceiverName> may refer to a name within the current workstation; if so, and the station is listening, it will receive its own message.

<SenderNameNum> is the slot number associated with a name previously added to the sender's local name table. The number is used by NetBIOS to manage the transmission. Slot number 1 is always safe to use, since this slot is associated with the adapter's permanent name:

The message will be ignored if the target station does not have a <ReceiveDatagram> call pending when the datagram is sent.

If <Wait> is True, the NetBIOS call does not return until the message has been transmitted. The application can check the value in <N.RetCode> to determine whether the transmission was successful. (Whether the message was received is another matter; datagram senders are not notified whether a given message was received, unless the receiver explicitly sends a reply. For messages requiring confirmation, use the NetBIOS session services instead.)

If <Wait> is False, the call returns immediately. The calling application can go ahead with other work while waiting for the transmission to occur. In the meantime, the <NCB> variable must remain undisturbed in memory because the NetBIOS will eventually write status information to it.

To determine when the action is complete, the application must poll the <CmdComplete> field of the <NCB> as follows:

```
while N.CmdComplete = NbeCommandPending do begin  
    {Do other work}  
end;  
if N.RetCode <> 0 then  
    {Handle error};
```

The <NetBiosCmdCompleted> function may also be used for this purpose. See <ReceiveDatagram> for an example.

SendDatagram

The untyped parameter <Datagram> may contain any data type. The maximum size of a datagram is implementation specific, but is typically 512 bytes. <DGSize> specifies the actual number of bytes to send.

A status code is returned in N.RetCode:

- \$00 Datagram sent.
- \$01 Invalid buffer length. (<DGSize> too large)
- \$13 Illegal name number. (sender name number doesn't exist)
- \$19 Name conflict detected. (duplicate names elsewhere on network)

See Also

ReceiveDatagram

SendBDatagram

C. SHARE: DOS Network Functions

The SHARE unit implements assorted network-related functions. Many will work whenever MS-DOS version 3.1 or later is in use. Others require loading the SHARE.EXE utility provided with the same versions of MS-DOS. Some require that the workstation be attached to an MS-NET compatible network such as 3Com's 3+. Novell's NetWare emulates most of these interfaces, so almost all the functions work with NetWare. We will point out the prerequisites for each routine. Often, we'll say that SHARE.EXE must be loaded, but in truth many network shells incorporate this function within another file.

Here is a list of the capabilities provided by the SHARE unit:

- determine whether SHARE is loaded
- lock and unlock any portion of a file
- set the retry count for locking errors
- flush DOS buffers to disk
- get DOS extended error information
- determine whether drive or file is on a remote machine
- determine or modify a printer setup string
- get current workstation name
- determine whether PC LAN is loaded
- get PC LAN version information
- redirect a network drive or printer to a local name
- cancel redirection

All of these concepts, with the possible exception of "redirection", will be clear to programmers. Redirection is used here in a different sense than typical for DOS, where it usually refers to channeling the standard input or output device to a different source or destination. In the context of MS-NET, redirection signifies the process of making a network resource (a printer or disk drive) available to a workstation. Redirection stores a name meaningful to the workstation in a table used by the MS-NET "redirector services", so that MS-NET can later translate a reference to the local name and redirect the resource request across the network. See <GetRedirectionEntry> and <RedirectDevice> for more information.

We recommend the following references for further information about the calls in the SHARE unit:

Advanced MS-DOS Programming, 2nd Edition

Ray Duncan, Microsoft Press, 1988.

DOS Programmer's Reference

Terry R. Dettman, Que Corporation, 1988.

PC Network Technical Reference

IBM manual #632205.

Most routines return standard DOS error codes. The SHARE unit defines three additional error codes, described in the Constants section below. For a list of DOS error codes, see the <GetExtendedError> function in this chapter.

Constants

PrinterSetupMax = 64;

Maximum length of a printer setup string.

WrongDosVersion = \$FFFD;

FileNotOpen = \$FFFE;

ShareNotLoaded = \$FFFF;

Non-DOS error codes returned by some SHARE functions.

Types

DeviceType = (DevPrinter, DevDrive);

Device types associated with redirection services.

NetworkStr = String[128];

String type used for the name of a network resource.

PClanOpType = (LanUnknown, LanRedirector, LanReceiver,
LanMessenger, LanServer);

Operating modes of an MS-NET or PC LAN workstation.

PrinterSetupStr = String[PrinterSetupMax];

String type used for printer setup strings.

Variables

DosMajor : Byte;

DosMinor : Byte;

The major and minor version numbers of MS-DOS. SHARE's initialization block sets these variables.

CancelRedirection

Declaration

```
function CancelRedirection(LocalName : NbNameStr) : Word;
```

Purpose

Cancels redirection previously established with <RedirectDevice>.

Description

<LocalName> must match the name originally specified. Requires DOS 3.1 or later, an MS-NET compatible network, and that SHARE.EXE be loaded.

A status code is returned in function result. Zero is returned for success; otherwise a DOS error code indicates the reason for failure. See

<GetExtendedError> for a list of error codes.

Example

```
if RedirectDevice(DevDrive, 'Q:', '\\MAIN\DBTEST', '', 2) = 0 then begin
  Writeln('Drive Q redirected to network');
  if CancelRedirection('Q:') = 0 then
    Writeln('Redirection cancelled for drive Q')
  else
    Writeln('Error cancelling redirection');
end else
  Writeln('Error redirecting drive Q');
```

Assigns drive Q: to the network resource \\MAIN\DBTEST, then cancels the redirection.

See Also

GetRedirectionEntry

RedirectDevice

DosLockRec

Declaration

```
function DosLockRec(var F; LockPosition, LockLength : LongInt) : Word;
```

Purpose

Lock part or all of a file using DOS services.

Description

Uses DOS call \$5C to lock all or part of a file. Requires DOS 3.0 or later, and that the DOS SHARE program be loaded.

The parameter <F> is untyped to allow any Turbo Pascal file variable to be passed. It is the caller's responsibility to assure that the variable passed is really a file. The file must already be open.

<LockPosition> specifies the start of the region to lock. The first byte of the file is at position 0. <LockLength> specifies the number of bytes to lock. It is not an error to lock beyond the end of the file. The same values of <LockPosition> and <LockLength> must be used when unlocking the file.

A DOS error code is returned in the function result: zero for success, else a code compatible with those listed under <GetExtendedError>.

Be sure to remove all locks before closing a file. Failure to do so will lead to unpredictable results.

Example

```
var
  F : file of Byte;
  B : Byte;
...
if DosLockRec(F, 5, 1) = 0 then begin
  Seek(F, 5);
  if IOResult <> 0 then begin
    Writeln('Seek error');
    if UnlockDosRec(F, 5, 1) <> 0 then ;
    Halt;
  end;
  B := 12;
  Write(F, B);
  if IOResult <> 0 then
    Writeln('Write error');
  if UnlockDosRec(F, 5, 1) <> 0 then
    Writeln('Error unlocking file');
end else
  Writeln('Error locking file');
```

Locks the sixth byte of file F, then seeks to that position and writes a new value to the byte there. (Note that the first byte is byte 0.) Note that the lock is removed even if the write to the file fails.

See Also

UnlockDosRec

Declaration

```
function GetExtendedError(var Class, Action, Locus : Byte) : Word;
```

Purpose

Returns extended information about the most recent DOS error.

Description

Requires DOS 3.0 or later. (Returns zeros for earlier versions of DOS.) <Class> is the type of the last DOS error (see Error Classes below). <Action> specifies the recommended action to take (see Recommended Actions below). <Locus> indicates the device that reported the error (see Error Locus Codes below). The function result returns the most recent DOS error code again. The table of extended error codes that follows may be used to interpret the error codes returned by other functions in the SHARE unit.

Extended Error Codes

\$01	Invalid DOS function number.
\$02	File not found.
\$03	Path not found.
\$04	Too many open files.
\$05	Access denied.
\$06	Invalid handle.
\$07	Memory control blocks destroyed.
\$08	Insufficient memory.
\$09	Invalid memory block address.
\$0A	Invalid environment.
\$0B	Invalid format.
\$0C	Invalid access code.
\$0D	Invalid data.
\$0E	Unknown unit.
\$0F	Invalid disk drive.
\$10	Attempted to remove current directory.
\$11	Not same device.
\$12	No more files.

(Critical error codes mapped by DOS)

\$13	Disk write-protected.
\$14	Unknown unit.
\$15	Drive not ready.
\$16	Unknown command.
\$17	Data CRC error.
\$18	Bad request structure length.
\$19	Seek error.
\$1A	Unknown media type.
\$1B	Sector not found.

GetExtendedError

\$1C	Printer out of paper.
\$1D	Write fault.
\$1E	Read fault.
\$1F	General failure.

(Codes new to DOS version 3)

\$20	Sharing violation.
\$21	Lock violation.
\$22	Invalid disk change.
\$23	FCB unavailable.
\$24	Sharing buffer exceeded.
\$32	Unsupported network request.
\$33	Remote machine not listening.
\$34	Duplicate name on network.
\$35	Network name not found.
\$36	Network busy.
\$37	Device no longer exists on network.
\$38	NetBIOS command limit exceeded.
\$39	Error in network adapter hardware.
\$3A	Incorrect response from network.
\$3B	Unexpected network error.
\$3C	Remote adapter incompatible.
\$3D	Print queue full.
\$3E	Not enough space for print file.
\$3F	Print file cancelled.
\$40	Network name deleted.
\$41	Network access denied.
\$42	Incorrect network device type.
\$43	Network name not found.
\$44	Network name limit exceeded.
\$45	NetBIOS session limit exceeded.
\$46	File sharing temporarily paused.
\$47	Network request not accepted.
\$48	Print or disk redirection paused.
\$50	File already exists.
\$52	Cannot make directory.
\$53	Fail on critical error.
\$54	Too many redirections.
\$55	Duplicate redirection.
\$56	Invalid password.
\$57	Invalid parameter.
\$58	Network device fault.
\$59	Function not supported by network.
\$5A	Required system component not installed.

Error Classes (<Class>)

\$01	Out of resource. (e.g., disk space or handles)
\$02	Temporary problem. (e.g., file locked)
\$03	Authorization problem.
\$04	Internal error in system software.
\$05	Hardware failure.
\$06	System software failure. (e.g., missing configuration file)
\$07	Application program error.
\$08	File or item not found.
\$09	Invalid type or format of file or item.
\$0A	File or item locked.
\$0B	Wrong or bad disk.
\$0C	Item already exists.
\$0D	Unknown error.

Recommended Actions (<Action>)

\$01	Retry reasonable number of times, then prompt to abort.
\$02	Retry reasonable number of times with delays between, then prompt to abort.
\$03	Get corrected information from user.
\$04	Clean up (e.g., release locks, close files) and abort application.
\$05	Abort application immediately.
\$06	Ignore error.
\$07	Retry after user intervention to correct error.

Error Locus Codes (<Locus>)

\$01	Unknown.
\$02	Block (disk) device.
\$03	Network.
\$04	Serial device.
\$05	Memory.

Example

```
if GetExtendedError(Class, Action, Locus) <> 0 then begin
  Write('Error location: ');
  case Locus of
    $01 : Writeln('Unknown');
    $02 : Writeln('Disk device');
    $03 : Writeln('Network');
    $04 : Writeln('Serial device');
    $05 : Writeln('Memory');
  end;
end;
```

Reports the locus of the last DOS error.

GetMachineName

Declaration

```
function GetMachineName(var MachName : NbNameStr;  
                        var MachNum : Byte) : Boolean;
```

Purpose

Returns the workstation's machine name and NetBIOS name index.

Description

Requires DOS 3.1 or later and an MS-NET compatible network.

<MachName> is the workstation's machine name. <MachNum> is the index into the machine's NetBIOS name table. There is no guarantee that <MachNum> will be unique for different workstations since it is just an index into the local NetBIOS name table. See Section 2.F for a discussion about how to determine unique workstation numbers for the FILER unit.

The function returns False only if DOS returns an error code. It may return True even if the machine has no defined name. In this case, <MachName> will be the empty string and <MachNum> will be zero.

Example

```
if GetMachineName(MachName, MachNum) then  
  Writeln('Workstation name: ', MachName)  
else  
  Writeln('GetMachineName function not supported');
```

Reports the name of the current workstation.

Declaration

```
function GetPrinterSetup(var SetupStr : PrinterSetupStr;  
    RedirIndex : Word) : Boolean;
```

Purpose

Returns the printer setup string for the specified device in the redirection table.

Description

Returns True if successful. Requires DOS 3.1 or later and an MS-NET compatible network.

<SetupStr> returns the current printer setup string. The printer setup string is sent prior to printing each file on the specified network printer. Thus, different stations can customize the operating mode of the printer.

<RedirIndex> is first established when <RedirectDevice> is called to redirect the local printer port to a network device. The first redirected resource obtains redirection index 0, the next index 1, and so on. If the index is not saved at the time of redirection, it may be obtained by calling <GetRedirectionEntry> for each index and comparing the returned information with the desired device.

This function returns False if <RedirIndex> doesn't specify a redirected printer or if the function is not supported.

Example

```
if GetPrinterSetup(SetupStr, 2) then  
  if SetPrinterSetup(NewSetupStr, 2) then begin  
    (Print a file)  
    ***  
    if not SetPrinterSetup(SetupStr, 2) then  
      Writeln('Unable to restore printer setup string');  
  end;
```

Saves the current printer setup string, activates a new setup string, prints a file, then restores the original setup.

See Also

GetRedirectionEntry
SetPrinterSetup

RedirectDevice

GetRedirectionEntry

Declaration

```
function GetRedirectionEntry(RedirIndex : Word;  
                             var LocalName : NbNameStr;  
                             var NetName : NetworkStr;  
                             var UserParam : Word;  
                             var Valid : Boolean;  
                             var Dev : DeviceType) : Word;
```

Purpose

Returns information about the specified redirection list entry.

Description

Requires DOS 3.1 or later, an MS-NET compatible network, and that SHARE.EXE be loaded.

<RedirIndex> is an index into the redirection table. The first redirection entry is number 0.

<GetRedirectionEntry> returns the other parameters. <LocalName> is a printer name like 'LPT1' or a drive name like 'G:'. <NetName> is the name of the network resource associated with the local name. <UserParam> is the same user-defined value assigned to the redirection entry when it was created. <Valid> returns True if this is a valid device. <Dev> returns either <DevPrinter> or <DevDrive>.

DOS error codes are returned in function result. See <GetExtendedError>.

Example

```
Index := 0;  
while GetRedirectionEntry(Index, LocalName, NetName,  
                           UserParam, Valid, Dev) = 0 do begin  
    if Valid then  
        Writeln(LocalName, ' is mapped to the network resource ', NetName);  
    inc(Index);  
end;
```

Reports on all redirected network resources.

See Also

CancelRedirection

RedirectDevice

GetTempFileName / IBMPCLANLoaded

Declaration

```
function GetTempFileName(PathN : PathName;  
    var TempFileName : PathName) : Word;
```

Purpose

Returns a file name guaranteed to be unique in the specified directory.

Description

Requires DOS 3.0 or later.

<PathN> is an existing drive and directory to hold the temporary file.
<TempFileName> returns the full pathname of the temporary file. The function result returns a DOS error code (see <GetExtendedError> for a list).

The file is created and immediately closed by this call. The application should refer to the returned name to rewrite or reset the file using an appropriate Turbo Pascal file variable.

Example

```
var  
    F : File;  
...  
    if GetTempFileName('C:\', TempFileName) = 0 then begin  
        Assign(F, TempFileName);  
        Rewrite(F, 1);  
    end;
```

Determines a unique file name in the root directory of drive C:, then opens it for writing.

Declaration

```
function IBMPCLANLoaded(var LanMode : PCLOpType) : Boolean;
```

Purpose

Returns True if IBM's PC LAN program is loaded.

Description

If PC LAN is loaded, then the parameter <LanMode> contains a value of type <PCLOpType>. This type indicates the mode in which the workstation is operating. Valid modes are <LanRedirector>, <LanReceiver>, <LanMessenger>, or <LanServer>. See the PC LAN documentation for more information on these operating modes. Novell's Advanced NetWare may pass the test for PC LAN. (It does so if Novell's INT2F TSR is loaded.) When determining the type of a network operating system, check first for Novell. See NETINFO.PAS for an example.

See Also

ShareInstalled

IsDriveLocal / IsFileLocal

Declaration

```
function IsDriveLocal(Drive : Byte) : Boolean;
```

Purpose

Determines whether the specified drive is local to the current workstation.

Description

Requires DOS 3.1 or later.

<Drive> is the number of the desired drive (0 = default, 1 = A, etc.). It is the application's responsibility to assure that <Drive> specifies a valid drive number. <IsDriveLocal> returns True if an invalid drive was specified. The recommended approach to determining whether a drive is valid is to switch to the drive by using DOS function \$0E and then to determine whether the current drive has been updated by using DOS function \$19. Turbo Professional and Object Professional provide the ValidDrive function that encapsulates these calls.

Example

```
Write('Current drive is ');  
if IsDriveLocal(0) then  
  Writeln('local')  
else  
  Writeln('a network drive');
```

Reports on the status of the default drive.

See Also

IsFileLocal

Declaration

```
function IsFileLocal(var F) : Boolean;
```

Purpose

Determines whether the specified file is local to the current workstation.

Description

Requires DOS 3.1 or later.

<F> must be a Turbo Pascal file variable of any type, already opened. <IsFileLocal> returns True when passed an invalid file variable.

Example

```
assign(F, 'TEST.DAT');  
reset(F);  
if IOResult <> 0 then Halt;  
if not IsFileLocal(F) then  
  Writeln('TEST.DAT is on a network drive');
```

Reports whether TEST.DAT is on a network drive.

Declaration

```
function RedirectDevice(TypeOfDev : DeviceType;  
    LocalName : NbNameStr;  
    NetWorkName : NetworkStr;  
    Password : NetworkStr;  
    UserParam : Word) : Word;
```

Purpose

Associates a local name with a network printer or disk.

Description

Requires DOS 3.1 or later, an MS-NET compatible network, and that SHARE.EXE be loaded.

<TypeOfDev> is either <DevPrinter> or <DevDrive> to specify printer or drive redirection, respectively.

For printer redirection, <LocalName> is one of 'PRN', 'LPT1', 'LPT2', or 'LPT3'. For drive redirection, <LocalName> is a drive designator such as 'F:'. After successful redirection, program references to these names will access network resources rather than local ones.

<NetworkName> specifies the name of the network resource. For example, to redirect the local drive F: to directory 'WORK' on an MS-NET server named 'MAIN', <NetworkName> should be set to '\\MAINWORK' and <LocalName> should be 'F:'. The syntax for specifying network directories may vary on different networks.

<Password> may be required by the network to gain access to the specified resource. Specify an empty string if no password is required.

<UserParam> is ignored by the network. Any value specified here will be returned by a call to <GetRedirectionEntry>, perhaps for use by the application.

The function result returns a DOS error code. See <GetExtendedError> for a list.

This function also works with Novell's Advanced NetWare. See NETINFO.PAS for an example.

It is good programming practice to restore the initial redirection settings when an application terminates. This is possible by using <GetRedirectionEntry> at program startup and then resetting any changed values with <RedirectDevice> at termination. Note, however, that <Password> values are never returned by <GetRedirectionEntry>, so to restore a password-protected resource requires that the application have prior knowledge of the password.

See Also

CancelRedirection

GetRedirectionEntry

SetPrinterSetup / ShareInstalled

Declaration

```
function SetPrinterSetup(SetupStr : PrinterSetupStr;  
    RedirIndex : Word) : Boolean;
```

Purpose

Defines a printer setup string for the specified device in the redirection table.

Description

Requires DOS 3.1 or later and an MS-NET compatible network.

<SetupStr> is the new printer setup string, which is sent prior to printing each file on the specified network printer. Thus, different stations can customize the operating mode of the printer.

<RedirIndex> is first established when <RedirectDevice> is called to redirect the local printer port to a network device. The first redirected resource obtains redirection index 0, the next index 1, and so on. If the index is not saved at the time of redirection, it may be obtained by calling <GetRedirectionEntry> for each index and comparing the returned information with the desired device.

Returns True if successful. This function returns False if the DOS version is too old, or if <RedirIndex> doesn't specify a redirected printer.

See Also

GetPrinterSetup

Declaration

```
function ShareInstalled : Boolean;
```

Purpose

Returns True if the DOS file-sharing module SHARE.EXE is installed.

Description

Requires DOS 3.0 or later.

See Also

IBMPCLanLoaded

Declaration

```
function UnlockDosRec(var F; FilePosition, FileLength : LongInt) : Word;
```

Purpose

Unlock all or part of a file using DOS services.

Description

Requires DOS 3.0 or later, and that the DOS SHARE program be loaded. The parameter <F> is untyped to allow any Turbo Pascal file variable to be passed. The variable passed must be an opened file. <LockPosition> specifies the start of the region to unlock and <LockLength> specifies the number of bytes to unlock. <LockPosition> and <LockLength> must exactly match the values used when the region was locked. You cannot, for example, specify

```
Status := UnlockDosRec(F, 0, FileSize(F));
```

to remove all locks at once. Be sure to remove all locks before closing a file. Failure to do so will lead to unpredictable results.

A DOS error code is returned in the function result. See <GetExtendedError> for a list.

See Also

DosLockRec

Declaration

```
function UpdateFile(var F) : Word;
```

Purpose

Flushes an open file to disk, assuring that previous writes are saved.

Description

Requires DOS 2.0 or later. The method used is to force a duplicate of the file handle, then close the duplicate. This is significantly faster than closing and reopening the file. The original file remains open after the call. For some networks, any locks that were in place before the call to <UpdateFile> are lost after the call. See the discussion of <IsamFlushDOS33> in Chapter 5 for more information.

The parameter <F> is untyped to allow any Turbo Pascal file variable to be passed. It is the caller's responsibility to assure that the variable passed is an open file. If the file is a Turbo Pascal text file, the application should call Turbo's Flush procedure first to assure that the Turbo text buffers are flushed as well.

Error codes are returned in the function result.

D. Network Demonstration Programs

Three demonstration programs are supplied with the multi-user version of B-Tree Filer. NETINFO can be run on any system; it attempts to determine what network is active and to report information about it. NSEND and NRECEIVE require two workstations logged onto a NetWare or NetBIOS-compatible network. NSEND copies files from one workstation to NRECEIVE on the other.

All three programs are command line driven. NETINFO accepts the following command line syntax:

NETINFO [Options]

At this time, the only option available is /P, which activates a detailed printer status report on NetWare installations. NETINFO can distinguish between the following network operating systems:

- NetWare
- NetWare with the NetBIOS emulator loaded
- NetBIOS
- PC LAN
- Other or no network

Because so many networks attempt to be compatible (or semi-compatible) with one another, it is possible that NETINFO may become confused when presented with an unknown network. Take a look at the logic used by the DetermineNetwork function in NETINFO.PAS for help in recognizing your target system.

NETINFO reports the most information when NetWare is detected. In this case, it writes out:

- server name and connection number
- server date/time
- whether transaction tracking is available
- workstation connection number
- workstation date/time
- NetWare version information
- workstation privileges, broadcast mode, and lock mode
- default network printer status
- whether print capture is activated
- printer setup strings for network printers

Since NetBIOS is a much lower level protocol, the only information NETINFO reports in this case is the workstation machine name and printer setup strings for any network printers.

In all cases, NETINFO reports the device redirection list, since this function can safely be called even without a network. (See <GetRedirectionEntry> in Section 7.C for more information about network redirection.)

NSEND and NRECEIVE work together to copy a file from one workstation directly to another, without using the server as an intermediary. They use either Novell SPX or NetBIOS session services to send the messages, depending on which is available.

If SPX is available, NSEND and NRECEIVE prompt to get the user name of the other workstation. This name is translated to a connection number, and then to an internet address for use with SPX.

If NetBIOS is used, NSEND and NRECEIVE define a common group name and then attempt to open a session. No prompting is required. Each program loops trying to make the connection so that synchronization isn't critical. If you need to abort either program before making the connection, just press <Esc>.

Regardless of the communication method, the sending program will prompt for a filename to transfer. The sender should specify the name of an existing file. The receiver will get a copy of this file in the current directory. NRECEIVE will warn if the copy would overwrite an existing file. Actually, the sender can use DOS wildcards to specify a group of files; each matching file will be copied to the receiver. The file is then copied directly from sender to receiver, without intervention of the server.

NSEND and NRECEIVE may also be called with command line parameters, which provide a few more options as well:

**NSEND [Options] [FileToSend [FileToSend]...]
NRECEIVE [Options]**

Options

/?	List the options
/A	Automatically find receiver (NetWare only)
/B	Force the use of NetBIOS services
/Rretries	Specify maximum number of connection retries (default 8)
/Username	Specify the other user name (NetWare only)
/V	Verbose: note each message block
/W	Force the use of NetWare SPX services

NSEND allows an arbitrary number of files on the command line. Each entry may contain DOS pathnames and wildcards. Prior to copying a file, NSEND sends a message to NRECEIVE with the filename to be copied. NRECEIVE strips off the directory name, if any, and checks to see whether the file exists in the current directory. If it does, NRECEIVE prompts the user whether to overwrite it. NRECEIVE then sends an acknowledgement message back to NSEND, telling it whether to send the file or not.

/B and /W are relevant only on systems (such as NetWare) that are capable of both NetBIOS and NetWare calls. /A and /U are mutually exclusive. /A tells NSEND to find another station by piping a message to all connections and listening for all replies; the first responding station becomes the target for the copy. /U just takes the place of the prompt for the other station's login name. /V enables a verbose mode where information about each block of a file transfer is displayed on the monitor. /R specifies the number of times the program will attempt to make a connection before aborting.

Here are a few examples of how to use NSEND and NRECEIVE with command line options:

```
NSEND /Upete *.PAS *.INC
NRECEIVE /Ujoe
```

Joe sends all PAS and INC files in the current directory to Pete, on a NetWare system.

```
NSEND /A *.PAS *.INC
NRECEIVE /A
```

On a NetWare system, transfer between any two cooperating stations.

```
NSEND /V /R20 C:\DOC\NOTICE.TXT
NRECEIVE
```

On a NetBIOS system, send C:\DOC\NOTICE.TXT to the station running NRECEIVE. Allow 20 retries to give the receiver time to set up, and use verbose mode to watch the whole file being transferred.

8. Appendix

A. Error Codes

The following table summarizes all error codes that can be returned in the <IsamError> variable of the FILER unit. See the READ.ME file for additional error codes that may have been added since the manual was printed. The format of the table is

ErrorNumber (ErrorClass)	Routines that may generate the error Brief description of the error
--------------------------	--

For more information about each error, see Chapter 5's detailed documentation about each corresponding routine. "General" errors occur in low level FILER procedures and cannot be correlated with a higher level interfaced procedure except from the context of the calling program. Also be aware of the interfaced variables <IsamDOSError> and <IsamDOSFunc>, which may be used to determine the cause of many low level DOS errors. See the description of these variables in Chapter 5.

8000 (4)	user The range 8000..8999 is reserved for user-defined codes, BONUS units, etc.
9500 (4)	general The range 9500..9899 is reserved for IOResult error codes. Subtract 9500 to get the true IOResult code, then refer to the Turbo Pascal Programmer's Guide or a DOS reference manual.
9900 (1)	general Invalid path name
9901 (4)	general Too many open files
9902 (4)	general Current directory is full
9903 (1)	general File not found
9904 (4)	general Invalid file descriptor
9905 (4)	general Read request exceeds 64K bytes
9906 (4)	general Write request exceeds 64K bytes
9907 (4)	general Error returning file size
9908 (4)	general Invalid file access mode
10000 (4)	BTInitIsam Number of page buffers is less than MaxHeight

- 10001 (3) BTAddRec
Serious I/O error in save mode. The fileblock is still in usable state but the last operation was not completed.
- 10002 (3) BTDeleteRec
Serious I/O error in save mode. The fileblock is still in usable state but the last operation was not completed.
- 10003 (3) BTAddKey
Serious I/O error in save mode. The fileblock is still in usable state but the last operation was not completed.
- 10004 (3) BTDeleteKey
Serious I/O error in save mode. The fileblock is still in usable state but the last operation was not completed.
- 10005 (3) BtDeleteAllKeys
Serious I/O error in save mode. The fileblock is still in usable state but the last operation was not completed.
- 10010 (4) BTOpenFileBlock
Index file probably corrupt (wasn't closed properly after last modification)
- 10020 (4) BTCreateFileBlock
Record length out of range (<21 bytes)
- 10030 (4) BTOpenFileBlock, BTCreateFileBlock
Insufficient memory for key descriptors
- 10050 (4) BTCreateFileBlock
Invalid number of keys specified (<0 or >MaxNrOfKeys)
- 10055 (4) BTCreateFileBlock
Key length out of range (<1 or >MaxKeyLen)
- 10060 (4) BTOpenFileBlock
Too many keys, or file read error (>MaxNrOfKeys)
- 10070 (4) general
File read error. Common reasons for this error include:
Dialog file left corrupted by a previous program run (delete the dialog file at the DOS command line);
BTGetRec specified an invalid record number (larger than BTFileLen);
two different programs compiled with different MaxNrOfWorkStations constants are trying to access the same fileblock;
a physical hard disk error.
- 10075 (4) general
File write error
- 10080 (4) BTCloseFileBlock
Fileblock not open
- 10090 (4) BTCreateFileBlock
Insufficient memory for an IsamFileBlock variable
- 10100 (4) BTOpenFileBlock
Insufficient memory for an IsamFileBlock variable
- 10110 (2) general
Drive not ready
- 10120 (4) BTOpenFileBlock
PageSize now incompatible with fileblock
- 10121 (4) BTOpenFileBlock
MaxKeyLen now incompatible with fileblock
- 10125 (4) BTAddKey
Key length too long (>max length for its index)

10130 (4)	BTPutRec Attempt to write to record number 0
10135 (4)	BTDeleteRec Attempt to delete record number 0
10140 (4)	general Unexpected DOS error (check IsamDOSError and IsamDOSFunc)
10150 (4)	general Out of handles on flush of network fileblock
10160 (4)	BTCloseFileBlock Fileblock not correctly closed
10164 (4)	general Invalid key number (<1 or >max index for this fileblock)
10170 (4)	BTAddRec, BTDeleteRec Free record list corrupt (attempt to use record 0)
10180 (4)	general Attempt to repair fileblock failed
10190 (4)	ExtendHandles Requires DOS 3.3 or later
10191 (4)	ExtendHandles Insufficient memory for new file handle table
10192 (4)	ExtendHandles Unable to obtain new file handle table from DOS
10200 (1)	BTFindKey No matching key found
10205 (1)	GetRecReadOnly Data record is locked
10210 (1)	BTSearchKey No key found and no larger keys available
10220 (1)	BTDeleteKey Key to delete was not found
10230 (1)	BTAddKey Cannot add duplicate key
10240 (1)	BTNextDiffKey No larger key found
10245 (1)	BTPrevDiffKey No smaller key found
10250 (1)	BTNextKey No larger key found
10255 (1)	BTNextKey Sequential access not allowed
10260 (1)	BTPrevKey No smaller key found
10265 (1)	BTPrevKey Sequential access not allowed
10270 (1)	BTFindKeyAndRef No matching key and record number found
10280 (1)	BTGetApprKeyAndRef Index empty

- 10285 (1) BTGetApprRelPos
Index empty
- 10303 (4) WSNrLogIn
Out of workstation numbers
- 10305 (2) WSNrLogIn
Unable to access log file
- 10310 (4) BTInitIsam
Network initialization error
- 10315 (4) BTExitIsam
Network exit error
- 10322 (4) BTCloseFileBlock
Attempt to remove readlock failed
- 10323 (4) BTCloseFileBlock
Attempt to remove record lock failed
- 10330 (2) BTLockFileBlock, BTLockAllOpenFileBlocks
Attempt to lock fileblock failed
- 10332 (2) BTReadLockFileBlock, BTReadLockAllOpenFileBlocks
Attempt to readlock fileblock failed
- 10335 (2) BTLockRec
Attempt to lock record failed
- 10337 (4) BTLockRec
Insufficient memory for lock list expansion
- 10340 (4) BTUnlockFileBlock, BTUnlockAllOpenFileBlocks
Attempt to unlock fileblock failed
- 10341 (4) general
Attempt to remove readlock failed
- 10342 (4) BTOpenFileBlock
Attempt to unlock fileblock failed
- 10345 (4) BTUnlockRec
Attempt to unlock record failed
- 10355 (2) BTOpenFileBlock
A lock prevents the operation
- 10356 (4) BTOpenFileBlock
Insufficient memory for network support record
- 10397 (2) general
Fileblock is read-only or a lock prevents operation
- 10398 (4) general
Fileblock must be locked for this operation
- 10399 (2) general
A lock prevents the operation
- 10410 (1) Reorg(V)FileBlock, Rebuild(V)FileBlock
Data file not found
- 10411 (4) Reorg(V)FileBlock, Rebuild(V)FileBlock
Insufficient memory for work buffer
- 10412 (4) Reorg(V)FileBlock, Rebuild(V)FileBlock
Section length exceeds 64K
- 10415 (4) general, VREC unit
Too many sections in a variable rec (total length exceeds \$FFF0)

- 10420 (4) BTGetApprKeyAndRef
Relative position or scale invalid
- 10425 (4) BTGetApprRelPos
Relative position or scale invalid
- 10430 (4) general
Fileblock repair not allowed in read-only mode
- 10435 (4) BTInitIsam
Index page larger than 16K bytes
- 10440 (4) BTOpenFileBlock
Cannot create dialog file in read-only mode
- 10445 (4) general
Fileblock pointer or descriptor corrupted
- 10446 (4) general
B-Tree Filer was called recursively from a user function such as a BuildKey routine. Such a recursive call is not allowed when EMS page buffers are used.
- 10447 (4) BTDestroyBufferContents
Attempt made to invalidate a modified index buffer
- 10450 (4) BTInitIsam
BTInitIsam called twice
- 10455 (4) general
BTInitIsam has not been called
- 10460 (4) Reorg(V)FileBlock, Rebuild(V)FileBlock
Reorganization aborted

B. Procedure and Function Reference

The unit name where the routines reside is listed next to each category. In each category, the routines are listed in alphabetical order.

Initialization and Fileblock Operations (FILER)

BTCloseAllFileBlocks

Close all currently opened fileblocks

BTCloseFileBlock

Close the specified fileblock

BTCreateFileBlock

Create a new fileblock

BTDeleteFileBlock

Delete a fileblock

BTExitIsam

Finish working with FILER

BTFlushAllFileBlocks

Store all new data of all open fileblocks

BTFlushFileBlock

Store all new data for a specified fileblock

BTForceNetBufferWriteThrough

Activate automatic flushing for all save mode, network fileblocks

BTForceWritingMark

Activate automatic flushing of the fileblock modified flag

BTInitIsam

Initialize network environment and allocate page buffers

BTIsamErrorClass

Divide <IsamError> into one of five error classes

BTNetSupported

Return the currently initialized network type

BTNoNetCompiled

Return True if FILER was compiled with the NoNet define

BTOpenFileBlock

Open an existing fileblock in specified mode

BTPeekNetSupported

Like **BTNetSupported** but leaves **IsamError** undisturbed

BTPeekNoNetCompiled

Like **BTNoNetCompiled** but leaves **IsamError** undisturbed

BTSetDosRetry

Set retry attempts and delay time for automatic DOS read retries

Data File Operations (FILER)

BTAddRec

Add a data record

BTDeleteRec

Delete a data record

BTGetRec

Read a data record with given reference number

BTGetRecordInfo

Return system information about a specified data record

BTGetRecReadOnly

Read a specified data record even if the record is locked

BTGetStartingLong

Read the first four bytes of a data record

BTPeekGetRecordInfo

Like **BTGetRecordInfo** but leaves **IsamError** undisturbed

BTPutRec

Write a modified record back to data file

Index File Operations (FILER)

BTAddKey

Add a key

BTClearKey

Reset the sequential pointer

BTDeleteAllKeys

Delete all keys in an index

BTDeleteKey

Delete a key

BTDestroyBufferContents

Mark index buffers invalid for specified fileblock

BTFindKey

Search for a specific key

BTFindKeyAndRef

Search for a key with a given data record reference

BTGetApprKeyAndRef

Return approximate key and reference for a given relative position

BTGetApprRelPos

Return approximate relative position for a given key and reference

BTKeyExists
Return True if a particular key exists

BTNextDiffKey
Search for the next different larger key

BTNextKey
Obtain the next key in ascending sequence

BTPrevDiffKey
Search for the next different smaller key

BTPrevKey
Obtain the next key in descending sequence

BTSearchKey
Search for a specific key

BTSearchKeyAndRef
Search for a key near the given key and data record reference

BTSetSearchForSequential
Modify the SearchForSequential option for a given index

Information about Fileblocks (FILER)

BTRecIsLocked
Return True if any record has been locked

BTDataFileName
Return the name of the fileblock's data file

BTDataRecordSize
Return the size of each data record

BTFileBlockIsLocked
Return True if fileblock is locked

BTFileBlockIsOpen
Return True if fileblock is open

BTFileBlockIsReadLocked
Return True if fileblock is read locked

BTFileLen
Determine the total number of records in the data file

BTFreeRecs
Determine the number of free data records

BTGetSearchForSequential
Return the SearchForSequential state for a given index

BTIndexFileName
Return the name of a fileblock's index file

BTIsNetFileBlock

Return True for a network fileblock

BTKeyRecordSize

Return the size of an index page block

BTMinimumDatKeys

Return the maximum record capacity of a drive

BTNrOfKeys

Return the number of indexes in a fileblock

BTOtherWSChangedKey

Return True if another workstation has modified a given index

BTPeekARecIsLocked

Like BTARecIsLocked but leaves IsamError undisturbed

BTPeekDataFileName

Like BTDataFileName but leaves IsamError undisturbed

BTPeekDatRecordSize

Like BTDatRecordSize but leaves IsamError undisturbed

BTPeekFileBlockIsLocked

Like BTFileBlockIsLocked but leaves IsamError undisturbed

BTPeekFileBlockIsOpen

Like BTFileBlockIsOpen but leaves IsamError undisturbed

BTPeekFileBlockIsReadLocked

Like BTFileBlockIsReadLocked but leaves IsamError undisturbed

BTPeekIndexFileName

Like BTIndexFileName but leaves IsamError undisturbed

BTPeekIsNetFileBlock

Like BTIsNetFileBlock but leaves IsamError undisturbed

BTPeekKeyRecordSize

Like BTKeyRecordSize but leaves IsamError undisturbed

BTPeekMinimumDatKeys

Like BTMinimumDatKeys but leaves IsamError undisturbed

BTPeekNrOfKeys

Like BTNrOfKeys but leaves IsamError undisturbed

BTPeekRecIsLocked

Like BTRecIsLocked but leaves IsamError undisturbed

BTRecIsLocked

Return True if given record is locked

BTUsedKeys

Return the number of keys added to an index

BTUsedRecs

Return the number of valid data records in data file

Locking Operations (FILER)

BTLockAllOpenFileBlocks

Reserve all open fileblocks for exclusive write access

BTLockFileBlock

Reserve a fileblock for exclusive write access

BTLockRec

Reserve a given data record for exclusive access

BTReadLockAllOpenFileBlocks

Prevent writing to any open fileblock by another workstation

BTReadLockFileBlock

Prevent writing to a fileblock by another workstation

BTUnlockAllOpenFileBlocks

Unlock all open fileblocks

BTUnlockAllRecs

Unlock all locked records of a fileblock

BTUnlockFileBlock

Unlock a fileblock

BTUnlockRec

Remove a lock from a specified record

EMS Heap Management (EMSHEAP)

EMSMaxAvail

Return the size of the largest available EMS block

EMSMemAvail

Return the total amount of free EMS space

ExitEMSHeap

Deinstall the EMSHEAP unit

FreeEMSMem

Free a previously allocated block of EMS memory

GetEMSMem

Allocate a memory block from the EMS heap

InitEMSHeap

Explicitly initialize the EMSHEAP unit

MapEMSPtr

Map a logical EMS pointer into a physical one

RestoreEMSCtxt

Restore a previously saved EMS context

SaveEMSCtxt

Save the current state of the EMS page frame

Variable Record Length Data File Operations (VREC)

BTAddVariableRec

Add variable length data record

BTCreateVariableRecBuffer

Allocate section buffer

BTDeleteVariableRec

Delete variable length data record

BTGetVariableRec

Read variable length data record

BTGetVariableRecLength

Return the total length of a variable record

BTGetVariableRecPart

Read part of a variable length record

BTGetVRecPartReadOnly

Read part of a variable length record even if locked

BTGetVRecReadOnly

Read a variable length record even if locked

BTPutVariableRec

Write a modified variable length record back to data file

BTReleaseVariableRecBuffer

Release section buffer

BTSetVariableRecBuffer

Allocate section buffer of specified size

Fileblock Browsing (BROWSER)

AddBrowseCommand

Modify browser keystroke mapping

Browse

Display and scroll through a fileblock

BrowseAgain

Return to previous location and browse

BrowseReadKey

Read keystroke for browser

DisableBrowseMouse

Disable mouse control of browser

DisableFiltering

Disable filtering function to display all records

EnableBrowseMouse

Enable mouse control of the browser

EnableFiltering

Specify function used to filter out undesired records

IsFilteringEnabled

Return True if filtering is enabled

RefreshAtEachCommand

Refresh function returns True if no keystrokes pending and index changed

RefreshPeriodically

Like RefreshAtEachCommand, but checks only at specified time intervals

Rebuilding Fileblocks (REBUILD, VREBUILD)

RebuildFileBlock

Purge deleted records and rebuild index file

RebuildVFileBlock

Rebuild fileblock containing variable length records

Reorganizing Fileblocks (REORG, VREORG)

ReorgFileBlock

Change record structure and rebuild index file

ReorgVFileBlock

Reorganize fileblock containing variable length records

Converting Fixed Fileblocks to Variable Length (FIXTOVAR)

FixToVarFileBlock

Change fixed length records to variable length and rebuild index file

Numeric Keys (NUMKEYS)

BcdToKey

Convert a BCD real to a string

DescendingKey

Invert string for a descending sort

ExtToKey

Convert an 8087 real to a string

IntToKey

Convert an integer to a string

KeyToBcd

Convert a string to a BCD real

KeyToExt

Convert a string to an extended

KeyToInt

Convert a string to an integer

KeyToLong

Convert a string to a longint

KeyToReal

Convert a string to a real

KeyToShort

Convert a string to a shortint

KeyToWord

Convert a string to a word

LongToKey

Convert a longint to a string

Pack4BitKey

Pack a suitable string into 4 bit characters

Pack5BitKeyUC

Convert suitable string to uppercase and pack into 5 bit characters

Pack6BitKey

Pack a suitable string into 6 bit characters

Pack6BitKeyUC

Convert suitable string to uppercase and pack into 6 bit characters

RealToKey

Convert a real to a string

ShortToKey

Convert a shortint to a string

Unpack4BitKey

Unpack string packed by Pack4BitKey

Unpack5BitKeyUC

Unpack string packed by Pack5BitKeyUC

Unpack6BitKey

Unpack string packed by Pack6BitKey

Unpack6BitKeyUC

Unpack string packed by Pack6BitKeyUC

WordToKey

Convert a word to a string

Miscellaneous Fileblock Utilities (ISAMTOOL)

ExtendHandles

Increase the number of available file handles

InvertString

Invert string for a descending sort (preserves length when called twice)

IsamErrorMessage

Return a text string for an error number

WSNrLogIn

Determine workstation number in hardware independent fashion

WSNrLogOut

Return workstation number to free pool

Sorting (MSORT)

AbortSort

Prematurely terminate sorting

DoSort

Low level sorting routine

AutoSort

High level sorting routine

AutoSortInfo

Return resource information for sort

PutElement

Pass a record into the sorting system

GetElement

Return a record from the sorting system

NetWare File Operations (NETWARE)

ConsolePriv

Determine whether the station has console privileges

FileIsShareable

Determine whether a file is shareable

GetDirHandle

Get the handle of a network drive

GetDirPath

Return the current directory associated with a directory handle

GetDirRights

Determine what rights the station has to use a directory

GetExtFAttr

Get an extended file attribute

GetServerDateTime

Get the current date and time from the server

GetServerInfo

Get version information from server

GetWSInfo

Get NetWare version information

IsLockModeExtended

Determine whether extended or compatibility mode locks are in use

MakeFileShareable

Make a file shareable

NetWareLoaded

Determine whether NetWare shell is loaded

SetExtFAttr

Set an extended file attribute

SetLockMode

Activate or deactivate extended locks

NetWare Transaction Tracking (NETWARE)

TTSAbort

End a transaction, assuring that none of it is written to disk

TTSAvailable

Determine whether transaction tracking services are available

TTSBegin

Begin a transaction

TTSDisable

Disable transaction tracking (network-wide)

TTSEnable

Enable transaction tracking (network-wide)

TTSend

End a transaction, committing it to disk

TTStatus

Determine whether transaction was successfully written to disk

NetWare Print Spooling (NETWARE)

CancelLPTCapture

Close capture file and erase it

EndLPTCapture

Close capture file and submit it for printing

FlushLPTCapture

Submit capture file for printing, and continue capture

GetBannerUser

Determine user name to appear on print job banners

GetDefaultLocalPrinter

Determine what local printer (LPT1, LPT2, LPT3) will be captured

GetPrinterStatus

Determine status of a server printer

GetPrintJobFlags

Get detailed status of a print job

LPTCaptureActive

Determine whether print capture is in progress

SetBannerUser

Set user name to appear on print job banners

SetDefaultLocalPrinter

Set the local printer to be captured

SetPrintJobFlags

Modify detailed status of a print job

SpecifyCaptureFile

Specify file where capture output will go

SpoolExistingFile

Submit an existing file to the print queue

StartLPTCapture

Start capturing local printer output

NetWare Basic Messaging (NETWARE)

BroadcastToConsole

Send a message to the server console

CheckMessagePipe

Determine what pipes are open

CloseMessagePipe

Close a pipe to a list of connections

GetBroadcastMode

Determine whether broadcast messages are received and displayed

GetBroadcastMsg

Read a pending broadcast message

GetConnFromName

Return the connection number of a login name

GetConnNo

Return the connection number of the station

GetDriverConfiguration

Return information about default server's LAN driver

GetInternetAddress

Return the physical address of a connection

GetMessagePipe

Read a pending piped message

LogNetworkMessage

Append a message to the server log file

OpenMessagePipe

Open a pipe to a list of connections

PreferredServerConnNo

Return connection ID of preferred server

PrimaryServerConnNo

Return connection ID of primary server

SendBroadcastMsg

Broadcast a message to a list of connections

SendMessagePipe

Send a piped message to a list of connections

ServerConnNo

Return the connection number of the server

SetBroadcastMode

Specify whether broadcast messages are received and displayed

SetPreferredConnNo

Specify the connection ID of the preferred server

NetWare Advanced Messaging (NETWARE)

IPXAssignUniqueSocket

Open and return a unique socket number

IPXCancelEvent

Cancel a send or listen event

IPXCloseSocket

Close a socket

IPXEventComplete

Determine whether a send or listen request is complete

IPXInternetAddress

Get the physical address of the current station

IPXListen

Submit a request to read an IPX message

IPXOpenSocket

Open a specified socket number

IPXRelinquish

Temporarily relinquish control to the IPX driver

IPXSend

Submit an IPX message to send

SPXAbortConn

Unilaterally abort a connection

SPXEventComplete

Determine whether an SPX request is complete

SPXListenCount

Determine how many listen buffers are active

SPXListenPooled

Determine whether an SPX message has been received

IPXServicesAvail

Determine whether IPX services are available

SPXEstablishConn

Submit a request to establish an SPX connection

SPXListenForConn

Submit a request to participate in an SPX connection

SPXReplenishAll

Reactivate all listen buffers

SPXReplenishPool

Reactivate a listen buffer

SPXSend

Submit an SPX message to send

SPXServicesAvail

Determine whether SPX services are available

SPXTerminateConn

Gracefully shut down a connection

NetBIOS Operations (NETBIOS)

CancelRequest

Cancel a pending NetBIOS request

ClearNCB

Clear a NetBIOS control block

NetBiosAddGroupName

Add a group name to the local name table

NetBiosAddName

Add a unique name to the local name table

NetBiosCall

Attempt to establish a session

NetBiosCallNoWait

Establish a session; no wait

NetBiosCmdCompleted

Determine whether no-wait command is now complete

NetBiosDeleteName

Remove a name from the local name table

NetBiosHangUp

Disconnect a NetBIOS session

NetBiosInstalled

Determine whether NetBIOS services are available

NetBiosListen

Listen for a new session request

NetBiosListenNoWait

Listen for session; no wait

NetBiosReceive

Receive a message via an opened session

NetBiosReceiveNoWait

Receive a session message; no wait

NetBiosRequest

Call the NetBIOS directly

NetBiosSend

Send a message via an opened session

NetBiosSendNoWait

Send a session message; no wait

ReceiveBDataGram

Receive a broadcast datagram

ReceiveDatagram

Receive a message using datagram services

ResetAdapter

Clear the name table and reset the hardware adapter

SendBDataGram

Broadcast a datagram message to all stations

SendDatagram

Send a message using datagram services

Share Operations (SHARE)

CancelRedirection

Cancel redirection of a local name

GetExtendedError

Get extended DOS error information

GetMachineName

Get MS-NET machine name

GetPrinterSetup

Get current setup string for a redirected printer

GetRedirectionEntry

Get information about a redirected name

GetTempFileName

Get unique file name

IBMPCLanLoaded

Determine whether PC LAN is loaded

IsDriveLocal

Determine whether a drive is local or on a remote machine

IsFileLocal

Determine whether a file is local or on a remote machine

LockDosRec

Lock part or all of a file

RedirectDevice

Redirect local name to network printer or drive

SetPrinterSetup

Set current setup string for a redirected printer

ShareInstalled

Determine whether SHAREEXE is installed

UnlockDosRec

Unlock part or all of a file

UpdateFile

Flush DOS buffers to disk

C. Converting from Borland's Database Toolbox

Because one of B-Tree Filer's main goals has been to remove the limitations of the Database Toolbox, we expect that many users of our package will be converting applications previously written to use the Toolbox. B-Tree Filer's programming interface and naming conventions are similar to those of the Toolbox, making conversion about as painless as possible. Nevertheless, B-Tree Filer's improvements could not occur without some change. This section will document the important differences. Of course, B-Tree Filer offers many added capabilities as well. To learn more about those, please refer to the prior reference sections of this manual.

We assume throughout this section that the first step in your conversion will be to convert to a single-user application using B-Tree Filer. We recommend this approach so that you can get your application up and running as quickly as possible with a minimum of debugging. Making this assumption also allows us to focus on the real differences between B-Tree Filer and the Database Toolbox and not get sidetracked by the new issues of multi-user programming. When you are ready for a multi-user conversion, see Chapter 4 for complete information about B-Tree Filer's multi-user capabilities.

For the remainder of this section, we will abbreviate Borland's Database Toolbox as "Dbox", and TurboPower's B-Tree Filer as "Filer". When referring to Dbox, we mean Borland's version for Turbo Pascal 4.0 and later, which provides some new capabilities compared to earlier versions. We will also refer to just the routines from Borland's low-level TACCESS.PAS unit; the higher-level unit TAHIGH.PAS will pose similar conversion questions.

There are three categories of differences that you'll need to consider while converting a Dbox application:

- routines with different names or calling conventions
- routines with different behavior
- data and index file formats

As a result of these differences, you'll need to make various changes to your source code and convert your data files to switch over to Filer. We'll discuss each of these in sections to follow.

We'll first offer one suggestion that probably goes without saying. Before you start your conversion, make a complete backup copy of your existing source and data files. In case something goes wrong, you'll feel much better if you have them.

Naming and Calling Conventions

Of course, the first change you'll need to make is to remove TACCESS from your USES statements and replace it with FILER. Be sure to replace it in all units--leaving even one reference to TACCESS in your program may lead to duplicate identifier names and the possibility for an endless variety of bugs that are very difficult to identify.

The names of Filer's routines are similar to those of Dbox, but Filer prefixes almost all of its routine names with the letters 'BT'. In some cases (for example, AddRec), you'll just need to add the prefix characters to get the Filer name (BTAddRec). In other cases, the names are quite different. The following list shows naming differences between Dbox and Filer that go beyond the initial prefix letters.

Dbox	Filer	Category
DataFile/IndexFile	IsamFileBlockPtr	Type
MakeFile/MakeIndex	BTCreateFileBlock	Procedure
OpenFile/OpenIndex	BTOpenFileBlock	Procedure
CloseFile/CloseIndex	BTCloseFileBlock	Procedure
FlushFile/FlushIndex	BTFlushFileBlock	Procedure
OK	IsamOK	Variable
TaStatus	IsamError	Variable

The first five differences stem from Filer's grouping of the data and index files as a single logical entity, the fileblock. You will replace each DataFile and all associated IndexFiles with a single <IsamFileBlockPtr>. Similarly, each call to MakeFile and all associated calls to MakeIndex will be replaced with a single call to <BTCreateFileBlock>. The same applies to each of the other file management routines. Filer allows 100 indexes or more for each fileblock, and uses just a single file handle to manage all of the indexes. Filer's use of a pointer to refer to each fileblock will also cut the use of global data space compared to Dbox.

The final two rows reflect our desire to make the identifiers names more explicit and self-consistent. If you don't want to rename all references to OK and TaStatus in your program, there's a fast fix. Just edit FILER.PAS and insert the following lines immediately after the declaration for <IsamError>:

```
OK : Boolean absolute IsamOK;  
TaStatus : Integer absolute IsamError;
```

Then your references to OK and TaStatus will get the values in Filer's variables. Remember, however, that the error codes returned in <IsamError> are completely different from those of <TaStatus>. See the next section for further details.

To minimize the manual effort required to perform name substitutions we've provided a program named UPGRADE.EXE which will automatically replace

Dbox's names with those of Filer. It also inserts comments in the source code to indicate where manual conversion efforts should be focused. A substitution file named DBOX.UPG is used to specify the exact replacements. DBOX.UPG is a simple text file; you can view or modify it with a standard text editor.

The UPGRADE program is described in more detail in Appendix D. The typical command line you'll use to upgrade a Dbox program is

UPGRADE /A /O OutputDir /S DBOX.UPG ProgramName

This says: upgrade all units of the program ProgramName, sending modified output to the directory OutputDir, using the substitution file DBOX.UPG.

Probably the most time-consuming portion of your conversion will result from the fact that Filer stores all indexes in a single file. Because of this, every Filer routine that deals with keys has an additional parameter that specifies which index is affected. Here is an example of one pair of corresponding routines.

Dbox

```
procedure AddKey(var IndexF : IndexFile;
                 var DataRef : LongInt;
                 var Key);
```

Filer

```
procedure BAddKey(IFBPtr : IsamFileBlockPtr;
                  Key : Word;
                  DataRef : LongInt;
                  Key : IsamKeyStr);
```

In making the conversion, you'll want to assign unique sequential numbers (starting with 1) to each IndexFile associated with a given DataFile. Then just replace the first parameter to each existing index routine with the new fileblock name followed by the number you've assigned to that index file. All of the following Dbox routines must be modified in this manner: AddKey, DeleteKey, FindKey, SearchKey, NextKey, PrevKey, and ClearKey.

The preceding procedure declarations point out another difference. Whereas Dbox uses an untyped VAR parameter for the key string, Filer specifies a typed value parameter. In this case, Filer's approach is less restrictive and should not require you to modify existing applications. For a few specific key routines, however, there are important behavioral differences that are described in the next section.

Another difference between Dbox and Filer lies on the border between syntactic and behavioral. Dbox requires the existence of an include file, TACCESS.DEF, which specifies various constants describing the configuration of the B-tree. Filer does not

use such an include file. The following table shows the constants specified by TACCESS.DEF and the action you should take for each one.

MaxDataRecSize

not used by Filer. Just refer to the SizeOf() your data record wherever a record length is required.

MaxKeyLen

edit FILER.CFG to specify the desired value in a constant of the same name.

PageSize

FILER.CFG defines such a constant having the value 62. We don't recommend that you change it, but you can if you wish. You can probably set Filer's constant to the same value you've been using for Dbox. Be sure to read the description of constant <PageSize> in Chapter 5 before changing FILER.CFG.

PageStackSize

not used by Filer. The index page buffer size of Filer is determined at runtime. This allows applications to take full advantage of available memory; with Filer, the page stack can buffer more than 64K bytes and use expanded (EMS) memory when available. See <BTInitIsam> in Chapter 5 for more information.

Order

not used by Filer. This constant is just one-half of PageSize.

MaxHeight

FILER.CFG defines such a constant having the value 8. We don't recommend that you change it, but you can if you wish. You can probably set Filer's constant to the same value you've been using for Dbox. Be sure to read the description of constant <MaxHeight> in Chapter 5 before changing FILER.CFG.

The reason we don't recommend that you change <PageSize> and <MaxHeight> is this: a frequent bug in Dbox applications stems from databases that grow beyond the size limit implicitly calculated from PageSize and MaxHeight. Since the overhead of using larger values for these constants is small, why not select them so you never need to worry about this problem? The default values in Filer allow up to 68 billion entries in the B-tree, with a minimum index file size of about 6K bytes and minimum page buffer heap space of about 20K bytes (when EMS is used, the minimum is reduced to just a few hundred bytes). If the space usage is a concern, the first constant you should consider changing is <MaxKeyLen>, which controls the overhead in a fairly linear fashion. If this isn't enough, change <PageSize> and <MaxHeight> only with great caution.

Toolbox Behavior

In certain situations, there are differences in the behavior of Dbox and Filer. We'll attempt to cover these one by one in the order you would encounter them as you trace the logical flow of your program. This list summarizes the more detailed discussion to follow:

- Filer's minimum data record size is 21 bytes compared to Dbox's 18.
- Filer uses a single fileblock to describe the Dbox DataFile and related IndexFiles.
- FileBlocks are allocated on the heap, while DataFiles and IndexFiles are generally stored in the data segment.
- Filer requires a call to <BTInitIsam> to dynamically allocate page space; Dbox allocates space based on a constant.
- Filer never halts after errors; you must check <IsamOK> after each call to a Filer routine. Filer's error codes differ from Dbox's, but both agree in assigning zero to success.
- After you create a new database with Filer, you must explicitly open it. Dbox leaves the database open after creating it.
- When you open an existing database with Filer, you don't need to specify the record size or the keys again.
- Dbox modifies the key string passed as a parameter to FindKey; Filer doesn't.
- Dbox returns the associated record number when DeleteKey is called to delete a primary (non-duplicate) key; Filer doesn't.
- Dbox's EraseFile and EraseIndex routines take *open* file variables as parameters; Filer's <DeleteFileBlock> routine takes a DOS pathname and erases the data and index files, presumed to be closed.

The following paragraphs expand upon these topics.

Both Dbox and Filer require that you specify a record structure for the data stored in each data file. The minimum acceptable size for a Dbox record is 18 bytes; for Filer it is 21 bytes. If your record is smaller than 21 bytes, you'll need to pad it to make up the difference. (The lower limit is set by the information stored in the reserved record--number 0--of the data file. Filer simply needs to store more there to manage reliable multi-user databases.) The record size change will be handled automatically when you convert your data file, as described in the next section.

The biggest behavioral difference is Filer's concept of a fileblock, which groups the data file and all associated index files in one logical entity. A fileblock is similar to Dbox's high-level DataSet, but the fileblock does not limit the number and type of keys as does the DataSet. Converting to fileblocks requires the syntactic changes

already described. If the basic operations for managing your database (adding a record and all its keys, modifying a record, and so on) have been isolated in modular fashion, you'll have no trouble in making the IndexFile to fileblock transition. If not, you'll have more edits to do, but the difficulty of conversion will be no greater.

Filer makes more frequent use of pointers in its management of data. Whereas DataFile and IndexFile variables use space in the segment where they are declared (about 150 and 200 bytes, respectively), Filer's <IsamFileBlockPtr> consumes only four bytes where it is declared and allocates the actual storage space on the heap. This approach reduces the strain on the 64K data segment. If your application restricts the heap with the {\$M} compiler directive, you may find that you need to increase the maximum heap space during the conversion.

Although both Dbox and Filer store the B-tree page buffers on the heap, Filer dynamically allocates it while Dbox sets the size based on the constant PageStackSize. With Filer's approach, application performance improves automatically when more system memory is available. Filer's method also lets it allocate page buffers larger than 64K bytes, leading to even better buffering when plenty of memory is available.

Another important difference occurs in the way the two toolboxes handle errors. Dbox halts for many errors, and returns OK set False for others. Filer never halts as a result of an error. It consistently sets <IsamOK> to False, and initializes <IsamError> to a specific value no matter what error has occurred. Filer therefore puts more responsibility to check for errors on the programmer, but offers much more flexibility in reacting to errors. Dbox isn't very explicit about what situations lead to fatal errors; you should add a check of <IsamOK> after every Dbox call when you perform the conversion.

The Dbox manual doesn't say much about the TaStatus variable either. This variable holds a code that refers to the most recent error. Because Filer returns a completely different set of error codes in <IsamError>, you'll need to map the numbers accordingly wherever your program refers to TaStatus. See Appendix A for a complete list of Filer's error codes. Note that the value 0 does refer to success in both cases.

Because Filer doesn't halt after errors, there is no need for an error handling routine like Dbox's TAEErrorProc. You may want to move some of the contents of any existing error routine into a general purpose exit procedure. You should remove all references to the variable TAEErrorProc.

The remaining items refer to specific Dbox procedures.

OpenFile and OpenIndex both require that you respecify the data record length or key length. Filer's <BTOpenFileBlock> routine does not allow this, because the appropriate information is already stored in the data file header. The two main parameters to <BTOpenFileBlock> are the fileblock pointer variable, and the root name of the data and index files to open. Several additional boolean parameters to <BTOpenFileBlock> specify whether the fileblock is being opened for network operation, read-only mode, or a special safety mode called save mode. During initial conversion, just set all of these parameters to False.

The FindKey routine in Dbox passes the key to search for as a VAR parameter, and uses the specified variable as a buffer area. As a result, the variable passed will be overwritten when the search fails. Filer passes the key as a value parameter, which means that the caller's parameter is never modified. This difference will require no conversion activity unless your application relies on the modified value returned by FindKey when the search fails. If so, you should call Filer's <BTSearchKey> routine when the <FindKey> search fails; <BTSearchKey> does return the nearest key string to the caller.

The DeleteKey routine in Dbox returns the record number associated with the deleted key if the index file contained primary (non-duplicate) keys. <BTDeleteKey> in Filer does not return the record number. It expects you to have determined the record number through a previous find key operation.

The EraseFile and EraseIndex routines take as parameters an open DataFile or IndexFile, respectively. Filer's <BTDeleteFileBlock> requires that the associated fileblock already be closed. It takes a DOS pathname as a parameter, and adds the 'DAT' and 'IX' extensions to that name before deleting the files.

Data and Index File Formats

Although both Filer and Dbox store data records one immediately after the other in the data file, the formats differ in one important respect. Both reserve the very first record of the data file to hold system information, and the organization of that information differs for the two. Hence, you must convert your Dbox data files in order to use them with Filer.

Practically speaking, the data file conversion doesn't cost anything, because the indexes must be totally rebuilt anyway. Filer's method of storing all indexes in a single file means that the existing Dbox index files are useless. They will be rebuilt from scratch during the conversion.

Filer offers the perfect tool to perform the conversion--the REORG unit described in Section 6.E. We cannot write a general purpose utility to convert your files, because we don't know how your keys are defined. We can, however, give you a

useful template to show how to write your own conversion program in just a few lines of code. The Convert program that follows shows how. You'll also find Convert in the file CONVERT.PAS which is created when you install B-Tree Filer.

```

program Convert;
uses
  Filer, Reorg;
type
  DataRecord =
    record
      RecDeleted : LongInt;
      LastName : String[20];
      CustNum : String[10];
      {...}
    end;
  const
    DataFileName = 'MYDATA';
    NrOfKeys = 2;
  var
    IID : IsamIndDescr;
    Pages : LongInt;
    RecordsAdded : LongInt;
    RecordsKeyed : LongInt;

    { $F+ }
  function BuildKey(var DatS; KeyNr : Integer) : IsamKeyStr;
  begin
    if RecordsKeyed = 0 then
      Writeln;
    with DataRecord(DatS) do
      case KeyNr of
        1 : BuildKey := CustNum;
        2 : BuildKey := LastName;
        {...}
      end;
    {Keep status counter running}
    inc(RecordsKeyed);
    if RecordsKeyed and 15 = 0 then
      Write(^M, RecordsKeyed);
  end;
  function ChangeDat(var DatSold; var DatSNew; Len : Word) : Boolean;
  begin
    if LongInt(DatSold) = 0 then begin
      {Record hasn't been deleted}
      ChangeDat := True;
      DataRecord(DatSNew) := DataRecord(DatSold);
      {Keep status counter running}
      inc(RecordsAdded);
      if RecordsAdded and 15 = 0 then
        Write(^M, RecordsAdded);
    end else
      {Record is deleted, don't add it}
      ChangeDat := False;
  end;
  { $F- }

```

```

procedure InitIID;
begin
  {!! Specify each index type here}
  IID[1].KeyL := 10;           {Maximum length of key string}
  IID[1].AllowDupK := False;   {False for a primary key}
  IID[2].KeyL := 20;
  IID[2].AllowDupK := True;    {True for a secondary key}
end;

begin
  Pages := BInitIsam(NoNet, 10000, 0); {Allocate heap space for page buffers}
  if not IsamOK then begin
    Writeln('Not enough memory available');
    Halt;
  end;
  RecordsAdded := 0;
  RecordsKeyed := 0;
  InitIID;
  ReorgFileBlock(DataFileName, SizeOf(DataRecord), NrOfKeys,
    IID, SizeOf(DataRecord),
    @BuildKey, @ChangeDat);

  Writeln;
  if IsamOK then
    Writeln(RecordsAdded, ' records converted')
  else
    Writeln('Convert failed. IsamError = ', IsamError);
end.

```

The main things you'll need to add to the template are the declaration of your data record, the function to build each key string, and the initialized structure that describes each key. Lines you'll need to modify are indicated with double exclamation points in the listing.

If you need to use non-default values of <MaxKeyLen>, <PageSize>, or <MaxHeight>, be sure to modify FILER.CFG for the correct values before you compile Convert.

Once you've customized and compiled the Convert program, copy your existing DBOX data file onto a new file with the extension 'DAT' and a name consistent with the one you've specified in Convert. The REORG unit requires the 'DAT' extension; anyway, you'll want to keep your original files safely hidden away in case something goes wrong. Then just run Convert. It will report status information while it runs. When it's done, you have new data and index files ready to run with B-Tree Filer.

D. Converting from B-Tree Filer 5.0

This section describes the differences between B-Tree Filer version 5.0 and this version, 5.2. The major changes and enhancements can be summarized as follows:

Single-user and Network Routines Merged

All routines in the FILER and VREC units now automatically differentiate between single-user and network fileblocks. This makes it easier and cleaner to write applications that work either way, and also easier to write higher level routines and objects that manage fileblocks in a general fashion. Most of the routines in these units were renamed in order not to generate ambiguity within existing applications. All such routines now begin with the 'BT' prefix. We'll describe the process of converting your applications below.

EMS Support

The index buffers of B-Tree Filer can now be completely or partially allocated from expanded memory, thus promising higher performance and freeing more of the DOS 640K for other purposes. This is done via the new <BTInitIsam> function, which combines the functions of the older <InitNetIsam> and <GetPageStack> routines.

Read-only Modes

The new fileblock opening procedure, <BTOpenFileBlock>, accepts new parameters <ReadOnly> and <AllReadOnly>. When <ReadOnly> is True, the current workstation is prevented from modifying the fileblock, and the FILER unit itself won't attempt to change the fileblock when closing it. When <AllReadOnly> is True, no workstation may modify the fileblock; B-Tree Filer uses this flag to optimize its own page buffering. The first option is useful for managing fileblock security; the second one is especially useful for CD-ROM applications.

As a result of these changes, B-Tree Filer 5.2 is not source code compatible with earlier versions. For example, the AddRec and AddNetRec routines are now replaced with a single procedure, <BTAddRec>, which works like one or the other depending on how the fileblock was opened. We've provided two methods to upgrade your earlier applications.

ISCOMPAT and VRCOMPAT Units

ISCOMPAT is for FILER unit compatibility and VRCOMPAT is for VREC compatibility. By adding these to the Uses statements of your existing applications, your program should compile and run as before, with just a few exceptions. Although these units provide the easiest technique to get up and running with Filer

5.2, you don't get any of the advantages of the cleaner calling interface, so we don't generally recommend that you use these units when converting to this new version.

Some upgrade problems are not solved by using ISCOMPAT and VRCOMPAT. The following procedures have been moved to the new ISAMTOOL unit. Add ISAMTOOL to your Uses statement if you call any of these routines.

```
IsamErrorMessage
ExtendHandles
WSNrLogIn
WSNrLogOut
```

The following procedures have been moved to the PREALLOC unit, which is found in PREALLOC.LZH on the BONUS disk.

```
PreAllocateFileBlock
PreAllocateVRecFileBlock
```

The following variables are now in the ISCOMPAT unit, but their interpretation is somewhat different than in B-Tree Filer 5.06 (the only previous version in which they were available).

```
InternalDosError - set for every DOS call, not just 10140 error calls
InternalDosFunction - now word instead of byte, initialized for every call
```

The following typed constants are no longer supported.

```
AbortReorg
NextDontUseKey
```

Both of these affect the operation of the REBUILD, REORG, VREBUILD, and VREORG units. You can get the effect of AbortReorg by setting <IsamReXUserProcPtr> to the address of a routine that checks for a user request to stop the reorganization. This routine then simply sets <IsamOK> to False to abort the operation. There is no one-to-one replacement for NextDontUseKey. If you don't want to add keys for a given index during a rebuild, set <AddNullKeys> to False and have the BuildKey routine return an empty string for each record in that index. Or call <BTDeleteAllKeys> for that index after reorganization is complete.

A number of FILER's internal identifiers have been changed or removed from the interface section. It's unlikely that these will affect you, but here is a list of the identifiers you can no longer access:

```
const    DummyFillChar
procedure ExpandFileBlock
var      FuncIsamChangeAttrToShareable
var      FuncIsamExitNet
var      FuncIsamLockRecord
var      FuncIsamUnLockRecord
```

```

var      IsamDriveNotReadyError
var      IsamInitializedNet
var      IsamForceFlushOfMark
var      IsamLockError
type     IsamLockedListBlock
type     IsamLockedListBlockPtr
type     IsamLockedListElement
var      IsamNetEmu
var      IsamNetRequested
var      IsamNrOfWS
var      IsamOfBLPtr
var      IsamPageStackSize
var      IsamSR1Ptr
type     IsamStackRec
type     IsamStackRecPtr
const    LockedListBlockSize
const    NetFileAttr
const    NovellFileAttr

```

UPGRADE Program

UPGRADE is a small utility that applies multiple identifier name substitutions to Turbo Pascal programs or units. It reads a substitution file (FILER.UPG in this case) and then parses the specified program and makes replacements as appropriate. UPGRADE can thus handle automatically most of the rote work of moving to B-Tree Filer 5.2. It also inserts comments into the resulting source text where changes will require additional effort on your part. You should not use the ISCOMPAT and VRCOMPAT units if you're going to run UPGRADE on your application.

Call UPGRADE as follows:

UPGRADE [options] ProgramName[.PAS]

Options:

/A

Upgrade all units listed in Uses statement

/O OutputDir

Specify drive or directory to receive upgraded files

/S SubstFile

Specify substitution file (default FILER.UPG)

UPGRADE will automatically find FILER.UPG if it's in the current directory, in the directory where UPGRADE.EXE is found (DOS 3.0 or later), or on the DOS PATH. You can specify a different substitution file pathname with the /S option.

Specify the /A option if you wish to upgrade an entire program, as opposed to a single unit. UPGRADE will then process all units listed in the main program's Uses statement, as well as any units listed in the Interface Uses statement of any unit. It will not process units listed only in an Implementation Uses statement.

If you don't specify an output directory, UPGRADE will overwrite the original files with the modified output. The original files will be renamed with the extension .S00, unless there is already a file with that extension, in which case .S01, .S02, and so on, will be used. We recommend that you send the modified output file to a separate directory.

Note that UPGRADE substitutes without regard for normal Pascal scope rules. For example, if you happen to have declared a local variable named AddKey, UPGRADE will change its name to BTAddKey, even in your declaration. You can scan through FILER.UPG to see whether this effect will cause problems for you. Note that such substitutions don't affect the code generated by the compiler, but may affect the readability of the program.

Wherever UPGRADE knows that you'll need to make additional changes, it inserts a comment of the form:

```
{**new param ISOLock:Boolean**}
```

You can search for these comments and make the appropriate changes manually, or just recompile your program and let the compiler find remaining issues. In some cases, the comments are for your information only and do not require a change in order to compile the program. See FILER.UPG for a list of the comments. Note that FILER.UPG is just a text file; you can modify it to perform additional substitutions or use different comments.

Error Codes

If your application reacts to specific <IsamError> values, be sure to compare the codes you use to the values in Appendix A and update them accordingly. B-Tree Filer 5.2 now uses common error checking routines at the entry and exit points of almost all of its interfaced routines. As a result, many errors that previously generated differing codes for different procedures (e.g. 10165..10179, Invalid Key Number) now return the same code in all cases (10164 for this example).

The error class 99 (unknown error) was removed. A value of 4 (hard error) is now returned by <BTIsamErrorClass> in case of an unknown error.

Other Minor Changes

<BTCreateFileBlock> leaves the fileblock closed and unallocated on the heap, unlike <MakeFileBlock>.

The typed constant <SearchForSequentialDefault> now has a default value of True rather than False.

The maximum number of indexes per fileblock was reduced from 750 to 254. This new limit, which still exceeds practical requirements by a large margin, provides for a simpler and faster internal design.

In versions of B-Tree Filer prior to 5.05, index routines would automatically truncate key strings whose length exceeded the maximum for a given index. Now, <BTAddKey> returns error 10125 if the key is too long. <BTFindKey> (and related key searching routines) no longer truncate the specified search key; if the search key is longer than the key in the index file, an exact match will not be returned.

The MsNetMachName define used to activate code that used a hashing algorithm to determine workstation numbers from machine names. Now it takes a number directly from the end of the machine name, as described in Section 2.F.

Identifiers

A

AbortSort 280
AddBrowseCommand 233
AddNullKeys 84
AllNodes 351
AutoScaleMouse 225
AutoSort 281
AutoSortInfo 283

B

BcdToKey 260
BiggestDataItem 278
BKtype 232
BroadcastToConsole 329
Browse 235
BrowseAgain 237
BrowseHelpIndex 232
BrowseHelpPtr 232
BrowseKeyID 230
BrowseKeyMax 230
BrowseKeyPtr 232
BrowseKeySet 230
BrowseMouseEnabled 224
BrowseMousePage 224
BrowseReadKey 239
BTAddKey 96
BTAddRec 97
BTAddVariableRec 205
BTaRecIsLocked 98
BTClearKey 98
BTCloseAllFileBlocks 99
BTCloseFileBlock 100
BTCreateFileBlock 101
BTCreateVariableRecBuffer 206
BTDataFileName 103
BTDataRecordSize 103
BTDeleteAllKeys 104
BTDeleteFileBlock 105
BTDeleteKey 106
BTDeleteRec 108
BTDeleteVariableRec 207
BTDestroyBufferContents 109
BTExitIsam 110
BTFileBlockIsLocked 111
BTFileBlockIsOpen 112
BTFileBlockIsReadLocked 113
BTFileLen 114
BTFindKey 115
BTFindKeyAndRef 116
BTFlushAllFileBlocks 118
BTFlushFileBlock 119
BTForceNetBufferWriteThrough 120
BTForceWritingMark 121
BTFreeRecs 122
BTGetApprKeyAndRef 123
BTGetApprRelPos 125
BTGetRec 126
BTGetRecordInfo 127
BTGetRecReadOnly 128
BTGetSearchForSequential 129
BTGetStartingLong 130
BTGetVariableRec 208
BTGetVariableRecLength 209
BTGetVariableRecPart 210
BTGetVRecPartReadOnly 211
BTGetVRecReadOnly 212
BTIndexFileName 131
BTInitIsam 132
BTIsamErrorClass 136
BTIsamLockRecord 93
BTIsamUnlockRecord 93
BTIsNetFileBlock 137
BTKeyExists 138
BTKeyRecordSize 140
BTLockAllOpenFileBlocks 141
BTLockFileBlock 142
BTLockRec 144
BTMinimumDataKeys 145
BTNetSupported 146
BTNextDiffKey 147

BTNextKey 148
 BNoNetCompiled 150
 BNoOfKeys 151
 BOpenFileBlock 152
 BOtherWSChangedKey 154
 BPeekaRecIsLocked 95
 BPeekDataFileName 95
 BPeekDatRecordSize 95
 BPeekFileBlockIsLocked 95
 BPeekFileBlockIsOpen 95
 BPeekFileBlockIsReadLocked 95
 BPeekGetRecordInfo 95
 BPeekIndexFileName 95
 BPeekIsNetFileBlock 95
 BPeekKeyRecordSize 95
 BPeekMinimumDatKeys 95
 BPeekNetSupported 95
 BPeekNoNetCompiled 95
 BPeekNoOfKeys 95
 BPeekRecIsLocked 95
 BPrevDiffKey 155
 BPrevKey 156
 BPutRec 158
 BPutVariableRec 213
 BReadLockAllOpenFileBlocks 159
 BReadLockFileBlock 160
 BRecIsLocked 161
 BReleaseVariableRecBuffer 214
 BSearchKey 161
 BSearchKeyAndRef 162
 BSetDosRetry 163
 BSetSearchForSequential 164
 BSetVariableRecBuffer 215
 BUnlockAllOpenFileBlocks 166
 BUnlockAllRecs 167
 BUnlockFileBlock 167
 BUnlockRec 168
 BUsedKeys 169
 BUsedRecs 170
 BuildARow 240

C

CallNameType 384
 CancelLPTCapture 317
 CancelRedirection 405
 CancelRequest 385
 CheckMessagePipe 330
 ClearNCB 385
 CloseMessagePipe 330
 ConnectionList 328
 ConnInfoType 328
 ConsolePriv 292
 ConvertRecord 251

D

DatExtension 84
 DayOfTheWeek 290
 DefaultAdapterNum 382
 DescendingKey 259
 DeviceType 404
 DiaExtension 84
 DisableBrowseMouse 241
 DisableFiltering 241
 DisplayARow 242
 DoManualInitEMSHeap 187
 DosLockRec 406
 DosMajor 404
 DosMinor 404
 DoSort 284
 DriverInformation 290

E

EMSHardErrorFuncPtr 187
 EMSHeapErrorFuncPtr 187
 EMSHeapInitialized 188
 EMSMaxAvail 189
 EMSMemAvail 190
 EMSPointer 188
 EnableBrowseMouse 243
 EnableFiltering 243
 EndLPTCapture 317

ExitEMSHeap 191
ExtendHandles 267
ExtToKey 260

F

FileIsShareable 293
FixToVarFileBlock 255
FlushLPTCapture 318
FragmentDescriptor 352
FreeEMSMem 192

G

GetBannerUser 318
GetBroadcastMode 331
GetBroadcastMsg 331
GetConnFromName 332
GetConnInfo 333
GetConnNo 333
GetDefaultLocalPrinter 319
GetDirHandle 294
GetDirPath 295
GetDirRights 296
GetDriverConfiguration 297
GetElement 285
GetEMSMem 193
GetExtendedError 407
GetExtFAttr 298
GetInternetAddress 334
GetMachineName 410
GetMessagePipe 334
GetPrinterSetup 411
GetPrinterStatus 319
GetPrintJobFlags 320
GetRedirectionEntry 412
GetServerDateTime 300
GetServerInfo 301
GetTempFileName 413
GetWSInfo 301
GRunLength 279

H

HandlesToUseForAlloc 187
HelpForBrowse 230

I

IBMPCLanLoaded 413
InitEMSHeap 194
IntToKey 260
InvertString 269
IPXAddress 352
IPXAssignUniqueSocket 355
IPXCancelEvent 355
IPXCloseSocket 356
IPXECB 352
IPXEventComplete 357
IPXHeader 353
IPXInternetAddress 357
IPXListen 358
IPXOpenSocket 359
IPXRec 353
IPXRelinquish 360
IPXSend 361
IPXServicesAvail 362
IsamAssign 93
IsamBlockRead 93
IsamBlockReadRetLen 93
IsamBlockWrite 93
IsamClearOK 93
IsamClose 93
IsamCompiledNets 92
IsamCopyFile 93
IsamDelay 93
IsamDelayBetwLocks 84
IsamDelete 93
IsamDOSError 92
IsamDOSFunc 92
IsamError 91
IsamErrorMessage 270
IsamExists 94
IsamExtractFileNames 94
IsamFile 88

IsamFileBlockName 88
IsamFileBlockPtr 89
IsamFileName 89
IsamFileNameLen 85
IsamFlush 94
IsamFlushDOS33 85
IsamForceExtension 94
IsamGetBlock 94
IsamIndDescr 89
IsamKeyStr 89
IsamLongSeek 94
IsamLongSeekEOF 94
IsamOK 91
IsamPutBlock 94
IsamRename 94
IsamReset 94
IsamRetriesForLock 84
IsamRewrite 94
IsamReXUserProcPtr 91
IsamVRecBufSize 204
IsamWSNr 91
IsDriveLocal 414
IsFileLocal 414
IsFilteringEnabled 244
IsLockModeExtended 302
IxExtension 84

K

KeyToBcd 262
KeyToExt 262
KeyToInt 262
KeyToLong 262
KeyToReal 262
KeyToShort 262
KeyToWord 262

L

LastFileName 279
LogNetworkMessage 335
LongToKey 260
LPTCaptureActive 320

M

MakeFileShareable 303
MapEMSPtr 195
MaxCols 230
MaxECBPool 351
MaxEMSHeapPages 176
MaxEMSHeapPages 187
MaxHeapToUse 278
MaxHeight 86
MaxKeyLen 86
MaxNrOfKeys 87
MaxNrOfWorkStations 87
MaxRows 230
MaxVariableRecLength 204
MergeOrder 278
MinEMSHeapPages 187
MinimizeUseOfNormalHeap 87
MinRows 230
MouseDownMark 224
MouseUpMark 224
MouseX1 225
MouseX2 225
MsgExtension 84
MSortIOResult 279
MSortStatus 279

N

NbeCommandPending 382
NbNameMax 383
NbNameStr 384
NCB 384
NetBiosAddGroupName 386
NetBiosAddName 387
NetBiosCall 388
NetBiosCallNoWait 389
NetBiosDeleteName 390
NetBiosHangUp 391
NetBiosInstalled 391
NetBiosListen 392
NetBiosListenNoWait 393
NetBiosReceive 394

NetBiosReceiveNoWait 395
NetBiosRequest 395
NetBiosSend 396
NetBiosSendNoWait 396
NetSupportType 90
NetWareLoaded 303
NetworkStr 404
NoNetMode 231
NovDateType 328
NoWait 383
NwPermanent 290

O

OpenMessagePipe 335

P

Pack4BitKey 263
Pack5BitKeyUC 263
Pack6BitKey 263
Pack6BitKeyUC 263
PageSize 47
PathName 279
PCLanOpType 404
PhysicalNodeAddress 353
PoolSocketLongevity 351
PreferredServerConnNo 336
PrimaryServerConnNo 337
PrinterDevice 315
PrinterSetupMax 404
PrinterSetupStr 404
PrintJobType 316
PutElement 285

R

ReadDataRecord 231
RealToKey 260
RebuildFileBlock 248
RebuildVFileBlock 248
ReceiveBDatagram 397
ReceiveDatagram 398

RedirectDevice 415
RefreshFunc 231
RefreshPeriod 231
ReorgFileBlock 252
ReorgVFileBlock 252
ResetAdapter 400
RestoreEMSCtxt 196
RetriesOnLock 231
RowRec 232
RowsToJump 231

S

SaveEMSCtxt 196
SavExtension 84
ScrollBarAttr 224
ScrollBarAutoSize 225
ScrollBarCol 225
ScrollBarHit 225
ScrollBarUp 225
ScrollMark 224
ScrollVertChar 224
SearchForSequentialDefault 87
SendBDatagram 400
SendBroadcastMsg 337
SendDatagram 401
SendMessagePipe 338
ServerConnNo 339
ServerInformation 291
SetBannerUser 321
SetBroadcastMode 340
SetDefaultLocalPrinter 321
SetExtFAttr 304
SetLockMode 304
SetPreferredServerConn 341
SetPrinterSetup 416
SetPrintJobFlags 322
ShareInstalled 416
ShortToKey 260
SliderAttr 224
SpecialTask 244
SpecifyCaptureFile 322

SPXAbortConn 362
SPXEstablishConn 363
SPXEventComplete 365
SPXHeader 354
SPXListenCount 366
SPXListenForConn 367
SPXListenPooled 370
SPXRec 354
SPXReplenishAll 371
SPXReplenishPool 371
SPXRetryCount 351
SPXSend 372
SPXServicesAvail 373
SPXTerminateConn 373
StartLPTCapture 323
STemp 278

T

ToLetFreePages 187
TTSAabort 309
TTSAvail 309
TTSBgin 310
TTSDisable 311
TTSEnable 311
TTSEnd 312
TTSStatus 313

U

UnlockDosRec 417
Unpack4BitKey 265
Unpack5BitKeyUC 265
Unpack6BitKey 265
Unpack6BitKeyUC 265
UpdateFile 417
UseEMS 278
UseReadLock 232
UserMousePtr 224
UseScrollBar 225
UsingEMS 279

V

VersionStr 87

W

WordToKey 260
WSNrLogIn 271
WSNrLogOut 272

Index

0

3Com 3, 27, 403
8087 chip 261
{ \$F+ } compiler directive 236
{ \$N+ } compiler directive 260, 262

A

Add a network interface 40
Add a record 62
Add a memo field 256
ADDRESS.DAT 12
Allocating page buffers 59
Alloy NTNX 3
API 374
Arcnet 3
ASCIIZ keys 30
ASCIIZeroKeys 30, 257
Audit file 299
AUTOEXEC.BAT 38
Automatic fileblock repairs 81
Automatic retries 84
Automatic screen refresh 223

B

Background task hook 226
Balanced trees 46
Banner page 321
BigHeap 277
BIGSORT.EXE 7
Branches 45
Broadcast messages 289, 324
Broadcast mode 331, 340
BROWSER, mouse support 31, 224
BTDEFINE.INC 25, 40, 216
BTFILER.TXT 15
Bussinger, Scott 257
B-tree 45

B-Tree Isam 2
B-Tree Net 2

C

Capacity of B-Tree Filer 1
Capture file 314
CBISNet 27
Circumventing a lock 71
Closing a fileblock 61
Compatibility mode 302, 304
Compiler capacity 33
Compiler options 33
CompuServe 24
Conditional defines 31
CONFIG.SYS 55, 267
Configuring B-Tree Filer 25
Connection ID 339
Connection number 354
Console privileges 292, 311
Context-sensitive help hook 226
CONVERT.PAS 448
Converting from Borland's Database
Toolbox 441
Converting from B-Tree Filer 5.0 450
Converting to variable length records 254
Copy-protection 5
Corrupted fileblocks 81, 245
Corrupted log file 272
Creating a fileblock 59
Critical error handler 42
CRT, using 31
Cut/paste help 20

D

Data and index file operations 61
Data definition 58
Data entry 10
Data file, first record in 44, 48
Data integrity 305
Data orphan 52
Data records 58

Database, definition of 43
Database Toolbox 441
Databases 46
Datagram 375, 398, 401
Dbox 441
Deadlock 69, 72
DebugEMSHep 175
Degree 45
Deleted records 58, 64, 207
DesqView 28
Dettman, Terry R. 404
Dialog file 55
Directory handle 294
Directory rights 296
Diskettes 5
DOS 3.1 record locking 3
DOS error code variable 92
DOS function code variable 92
Duncan, Ray 404
Dynamic sockets 343

E

Editor Toolbox 226
EMS 174, 273, 277
EMS error codes 184
EMS for heap storage 173
EMSDisturbance 29
EMSHEAP 29
Enz EDV-Beratung GmbH 23
Error codes, FILER 421
Error codes, EMSHEAP 184
Error handling strategy 57
Error variables 91
Ethernet 3, 353
Event service routine 352, 369
Exclusive access 69
Expanded memory 174, 273
Explicit transactions 305
Extended attributes 304
Extended error codes, DOS 407
Extended file attributes 298

Extended lock mode 302

F

Far model 236
Fast indexing 298
Fast Update Plan 5
FASTUPD.xxx 5
FBROWSE.LZH 8
File and directory attributes 290, 293
File buffer flushing 85
File extensions 84
File format, variable length 202
File handles 267
File name length, maximum 85
File names 84
File servers 288
Fileblock, definition of 43, 55
Fileblock maintenance 203
Fileblock name 88
Fileblock pointer 89
Filer 5.0, converting from 450
FILER.MAK 7
Filtering records, BROWSER 220
First record in data file 44
Flushing files 85
Functions, short descriptions 426

G

Group name 374, 386
Groupware 287

H

Handles, file 267
Handles per fileblock 82
Haugdahl, J. Scott 376
Heap management 132
Heap, using EMS 173
Heap6 32
Height, B-tree 45, 86
Help hook, BROWSER 226

Help, POPHELP 15
Hidden swap files 16
Hot key, POPHELP 16
Hypertext help 5

I

Implicit transactions 305
Index description 89
Index file 44, 48
Index file size 102
Index keys 45, 48, 257
Indexed search 66
InitAllUnits 31
Installable keyboard hook 226
Installation 5
Integrity of a database 245
Internetwork Packet eXchange protocol 342
IPX 289, 342
ISAMTOOL 266

J

Jordan, Larry 287

K

Key assignment, changing 233
Key length, maximum 86, 89
Key types 30
Keyboard handling 226
Keys 44
Keys, maximum number 87

L

Lantastic 27
Leaf node 50, 51
Leaves 45
LengthByteKeys 30
Levels of locking 69
LHARC.EXE 6
LHARCMAN.LZH 6
Local name table 374

Locking 70
Logical record counts 198

M

Macro Assembler 3
Madrone, T.W. 287
MAKE 7
MAKEHELP.EXE 16
ManualInitEMSHeap 175
Mapping EMS context, MSORT 277
Mapping EMS pointers 180
MASM 3
Maximum file name length 85
Maximum length of a key 49
Memo fields 197, 256
Memory, out of 33
Merge order 278
Merge sort 273
Message pipes 324
Minimum data record size 445
Modifying records 64
Mouse support, BROWSER 224
MS-NET 3, 27, 403
MsNet 26
MsNetMachName 27

N

Nagel, Ralf 23
NCB 375
NetBIOS 374
NetBIOS Control Block 375, 384
NetBIOS name table 386, 387
NETDEMO 8, 10
NETFILER.MAK 7
NETINFO 8, 287, 418
NetWare 324, 335, 403
NetWare API 288
NetWare shell 303
Network demonstration programs 418
Network emulation 82
Network interfaces, adding 40

Network prerequisites 83
Network-OS 3, 27, 288
No-wait option 379
Node, B-tree 45, 47
Node merging and splitting 52
NoErrorCheckEMSHeap 175
Non-indexed search 67
NoNet 26
Novell NetWare 3, 26, 288
NRECEIVE/NSEND 287, 419
NTNXNet 27
Numeric keys 96, 257
NUMKEYS 48

O

Object Professional 8, 216
Object-oriented browser 216
OPCRT, using 31
Opening a fileblock 60
Order of a B-tree 46
Out of memory 33
Overlaying B-Tree Filer units 33
Overlays and EMS 177

P

Packed key strings 257, 263
Packets 342
PACKING.LST 6
Page balancing 52
Page buffers 57, 59
Page size 87
Password 415
Pasting help text 20
PC-LAN 3, 27, 379, 413
PC-MOS/386 27
PC-NET 3, 27
Permanent node name 374
Pipes 289
Pool index number 354
POPHELP 15
Post-event routine, NetBIOS 381

Primary keys 11, 48, 60
Print job 316
Print job flags 320
Print spooling 288, 314
PRINTDOC.EXE 6
Procedures, short descriptions 426
Purchase Agreement 23
Purge 245

Q

QuickPascal 3

R

Random access 44
Reacting to locks 76
Reacting to modified data 77
Read locks 70
READ.ME 5
READ.MEI 5
Record filtering 220
Record numbers 44
Record validation 221
Redirection 403, 415
Refresh function 223
Removing data 63
Reorganizing a fileblock 250
Requirements 3
Retries 84
Root node 45, 50
Run buffer, MSORT 277
Runtime fees 23

S

Save mode 10, 55, 81
Scan codes 19
Schatt, Stan 287
Schwaderer, W. David 376
Screen refresh 223
Scroll bar 224
Search paths 7

Search tree 46
Searching for data records 66
Secondary keys 48, 60
Section length 197
Sections of variable length records 14, 197
Sequenced Packet Exchange Protocol 344
Sequential access pointer 50, 65, 164
Sequential data access 44
Server crashes 245
Session, NetBIOS 375, 392
SFT 305
SHARE 26, 287, 403
Shareable files 293, 303
SHELL.CFG 267, 298
Shift key codes 18
SIMPDEMO.PAS 11
Site licensing 23
Socket numbers 343
Sorting B-Tree Filer fileblocks 275
Special task, BROWSER 230, 236, 244
SPX 289, 344
System record 102

T

TACCESS 441
TASM 3
Telephone support 24
Temporary index 220
TPALLOC 276
TPC.CFG 7
Transaction, definition of 310
Transaction tracking 81, 288, 298, 305
Tree structure 45
TSR swapping 15
TTS 298, 305
Turbo Assembler 3
Turbo Pascal 3
Turbo Professional 10, 216, 226, 228

U

Underflow 52

Unique names, NetBIOS 374, 387
UPGRADE.EXE 6, 442
UseEMSHeap 29
UseOPCRT 31, 216
UseOPEMS 175
UseTPCRT 31, 216
UseTPEMS 175

V

Validation function 221
Variable length records 10, 197, 235
Version information, file server 301
Version string 87
VinesNet 27
VREC 197

W

Warranty 24
Well-known sockets 343
WordStar 226, 233
Workstation numbers 3, 36, 91, 271
Workstations, maximum number 87

X

Xerox 342, 344
XMS memory 17

B-Tree Filer™

Turbo Pascal Tools for Powerful Multi-User Databases

Write powerful multi-user databases faster and easier using B-Tree Filer and Turbo Pascal for Windows or DOS. Convert your applications that use Borland's Database Toolbox to multi-user capability. B-Tree Filer is available in single-user and network versions. The single-user version of B-Tree Filer is upwardly compatible with the network version but does not contain network support code. The network version may be used for single-user applications.

The Performance and Connections You Need

You'll have the fastest, safest, most flexible databases when you use B-Tree Filer – no rigid structure, no TSR hassles, no running out of files. And your databases are compatible with Novell, Lantastic, MS-NET, and others.

The Complete B-tree Tool Kit

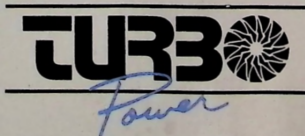
You get over 200 Turbo Pascal procedures that give you features like:

- Two billion records per database
- Record size 21 to 65,535 bytes
- Fixed or variable length records
- B+tree indexing with advanced page balancing
- 100 indexes per index file
- EMS support for index buffers
- Automatic journaling with automatic rebuild after a crash
- Support for single user, Novell, Lantastic, MS-Net, Vines, NTNX, Network OS, PC-MOS 386, DesqView, and DOS 3.X generic
- Up to 32,768 workstations per network
- Read only mode especially for CD-ROM databases
- Units for sorting, reindexing, reorganizing and browsing
- Units for network services like print spooling, messaging, and more

Full Source Code, Support, and No Royalties

You'll gain the confidence and flexibility of having complete source code. B-Tree Filer also includes full documentation, pop-up help, and plenty of demo programs. Technical support is provided on CompuServe and by telephone. You pay NO royalties.

B-Tree Filer requires Turbo Pascal for Windows or Turbo Pascal 6.0, 5.5, or 5.0. Also requires PC-DOS or MS-DOS 2.0 or later, and an IBM PC, XT, AT, PS/2 or compatible.



TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009
CompuServe I.D. 76004, 2611

©TurboPower Software 1991